ZERTIFIZIERENDE VERTEILTE ALGORITHMEN

KIM VÖLLINGER

Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät
Humboldt-Universität zu Berlin

Präsidentin der Universität:
Prof. Dr. Dr. Sabine Kunst

Dekan der Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:
Prof. Dr. Wolfgang Reisig
Prof. Dr. Kurt Mehlhorn
Prof. Dr. Holger Schlingloff

Verteidigt am 8.6.2020

ABSTRACT

A major problem in software engineering is to ensure the correctness of software. While *testing* is important in practice, it offers no mathematical correctness. *Formal verification* guarantees complete correctness but is often too costly. *Runtime verification* stands between the two methods as a often less costly method that offers mathematical correctness. Runtime verification answers the question whether an input-output pair is correct. A *certifying algorithm* convinces its user at runtime by offering a correctness argument. This is especially appealing for outsourced computations where the user has no insight on the algorithm and does not want to trust the developers blindly.

For each input, a certifying algorithm computes an output and additionally a *witness* – a correctness argument for the input-output pair. For example, an odd cycle in a graph G is a witness for G not being bipartite. Each certifying algorithm has a *witness predicate* – a predicate with the property: being satisfied for an input, output and witness implies the input-output pair is correct. A simple algorithm deciding the witness predicate for the user is a *checker*. Hence, the checker's correctness is crucial to the approach and motivates *formal instance verification* where we verify checkers and obtain machine-checked proofs for the correctness of an input-output pair at runtime. Certifying *sequential* algorithms are well-established with a theory, case studies, design patterns and a a framework for formal instance verification [McC+11; Riz15].

Distributed algorithms, designed to run on distributed systems, behave fundamentally different from sequential algorithms: their output is distributed over the system or they even run continuously. We investigate *certifying distributed algorithms*. Our research question is: How can we transfer the concept of certifying *sequential* algorithms to *distributed* algorithms such that we are in line with the original concept but also adapt to the conditions of distributed systems? In this thesis, we present a method to transfer the concept: Weighing up both sometimes conflicting goals, we develop a class of certifying distributed algorithms that compute distributed witnesses and have distributed checkers. We offer case studies, design patterns and a framework for formal instance verification. Additionally, we investigate other methods to transfer the concept of certifying algorithms to distributed algorithms.

ZUSAMMENFASSUNG

Eine Herausforderung der Softwareentwicklung ist, die Korrektheit einer Software sicherzustellen. Während *Testen* wichtig in der Praxis ist, bietet es keine mathematische Korrektheit. *Formale Verifikation* garantiert zwar vollständige Korrektheit, ist jedoch oft zu aufwändig. *Laufzeitverifikation* steht als weniger aufwändige Methode, die mathematische Sicherheit bietet, zwischen den beiden Methoden. Laufzeitverifikation beantwortet die Frage, ob ein Eingabe-Ausgabe-Paar korrekt ist. Ein *zertifizierender Algorithmus* überzeugt seinen Nutzer durch ein Korrektheitsargument zur Laufzeit. Insbesondere interessant ist das für ausgelagerte Berechnungen, bei denen ein Nutzer keinen Zugriff auf den Algorithmus hat und diesem auch nicht blind vertraut.

Dafür berechnet ein zertifizierender Algorithmus für eine Eingabe zusätzlich zur Ausgabe noch einen *Zeugen* – ein Korrektheitsargument für das Eingabe-Ausgabe-Paar. Beispielsweise ist ein ungerader Kreis in einem Graphen G ein Zeuge dafür, dass G nicht bipartit ist. Jeder zertifizierende Algorithmus besitzt ein *Zeugenprädikat*: Ist dieses erfüllt für eine Eingabe, eine Ausgabe und einen Zeugen, so ist das Eingabe-Ausgabe-Paar korrekt. Ein simpler Algorithmus, der das Zeugenprädikat für den Nutzer entscheidet, ist ein *Checker*. Die Korrektheit des Checkers ist folglich notwendig für den Ansatz und die *formale Instanzverifikation*, bei der wir Checker verifizieren und einen maschinengeprüften Beweis für die Korrektheit eines Eingabe-Ausgabe-Paars zur Laufzeit gewinnen. Zertifizierende *sequentielle* Algorithmen sind gut untersucht. Es gibt eine Theorie, Fallstudien, Entwurfsmuster, sowie ein Framework für die formale Instanzverifikation [McC+11; Riz15].

Verteilte Algorithmen, die auf verteilten Systemen laufen, unterscheiden sich grundlegend von sequentiellen Algorithmen: die Ausgabe ist über das System verteilt oder der Algorithmus läuft fortwährend. Wir untersuchen *zertifizierende verteilte Algorithmen*. Unsere Forschungsfrage ist: Wie können wir das Konzept zertifizierender *sequentieller* Algorithmen so auf *verteilte* Algorithmen übertragen, dass wir einerseits nah am ursprünglichen Konzept bleiben und andererseits die Gegebenheiten verteilter Systeme berücksichtigen? Wir stellen eine Methode der Übertragung vor. Die beiden Ziele abwägend entwickeln wir eine Klasse zertifizierender verteilter Algorithmen, die verteilte Zeugen berechnen und verteilte Checker besitzen. Wir präsentieren Fallstudien, Entwurfsmuster und ein Framework zur formalen Instanzverifikation. Darüber hinaus erläutern wir weitere Methoden der Übertragung.

DANKSAGUNG

Ich bedanke mich bei allen, die mich unterstützt haben. Besonderer Dank gebührt meinem Betreuer Prof. Dr. Wolfgang Reisig, der mir an seinem Lehrstuhl eine ideale Arbeitsumgebung bot. Ich danke Prof. Dr. Kurt Mehlhorn, der sich die Zeit nahm, mit mir erste Richtungen zu formulieren. Ich bedanke mich bei Prof. Dr. Holger Schlingloff, der mir im Rahmen des CrESt-Projekts eine industrielle Fallstudie ermöglichte.

Ich danke meinen Kolleg:innen: Birgit Heene, Andre Moelle, Robert Prüfer, Dr. Jan Sürmeli und Dr. Marvin Triebel haben meine Zeit am Lehrstuhl bereichert. Besonderer Dank gebührt den Student:innen, deren Abschlussarbeiten ich betreute: Samira Akili, David Asher und Alexander Boll. Samira und Alex danke ich auch für das Korrekturlesen.

Ich danke dem Graduiertenkolleg SOAMED, das mir als Assoziierte ein Netzwerk für den wissenschaftlichen Austausch bot. Insbesondere danke ich Prof. Dr. Uwe Nestmann und seinem Lehrstuhl, die mich für Diskussionen empfingen. Auch Prof. Dr. Sabine Glesner und ihr Lehrstuhl boten mir die in einer frühen Phase Raum für Diskussion, genau wie Prof. Dr. Holger Giese in einer späteren Phase. Insbesondere die Diskussionen mit Lucas Sakizloglou über Laufzeitverifikation im Allgemeinen waren hilfreich. Darüber hinaus führten Gespräche mit Prof. Dr. Björn Scheuermann zur Entwicklung einer Simulationsumgebung für zertifizierende verteilte Algorithmen. Dank gebührt auch Dr. Jens Gerlach und seiner Verifikationsgruppe am FOKUS. Ich lernte dort nicht nur als Studentin den Beweisassistenten COQ kennen, sondern wurde darüber hinaus auch durch Feedback zu meiner weiteren Forschung unterstützt.

Zu guter Letzt danke ich Familie und Freund:innen für die Unterstützung jeglicher Art. Insbesondere danke ich meiner Mutter Dr. Ulrike Völlinger, die meine Liebe zur Mathematik, Logik und Informatik weckte. Ebenso danke ich meinem Ziehvater Patrick Munck, der mich zur Gelassenheit erzog und mir somit half unter Stress einen klaren Kopf zu bewahren. Abschließend danke ich meinem Partner Armando Alibrandi, der mich in der Endphase dieser Arbeit bekochte und für den nötigen Ausgleich sorgte.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

CODE-BLOCK-VERZEICHNIS

Teil I

ÜBER DIESE ARBEIT

In diesem Teil stellen wir die vorliegende Arbeit kurz vor. Die Forschungsfrage, die wir betrachten, ist die folgende:

> Wie können wir das Konzept zertifizierender *sequentieller* Algorithmen so auf *verteilte* Algorithmen übertragen, dass wir einerseits nah am ursprünglichen Konzept bleiben und andererseits die Gegebenheiten verteilter Systeme berücksichtigen?

In Kapitel 1 motivieren wir unsere Forschungsfrage und erläutern unsere Ziele bei der Übertragung des Konzepts auf verteilte Algorithmen.

In Kapitel 2 geben wir einen Überblick über die Ergebnisse, die wir im Rahmen der vorliegenden Arbeit gewonnen haben, und erläutern den Aufbau dieser Arbeit.

1 MOTIVATION

In Abschnitt 1.1 legen wir in Kürze die nötigen Grundlagen, um unsere Forschungsfrage zu erläutern. In Abschnitt 1.2 motivieren wir unsere Forschungsfrage und erläutern unsere Ziele, die wir dabei verfolgen, sowie bestehende Herausforderungen.

1.1 ZERTIFIZIERENDE ALGORITHMEN & VERTEILTE ALGORITHMEN IN KÜRZE

Wir führen zertifizierende sequentielle Algorithmen ein (Abschnitt 1.1.1), sowie verteilte Algorithmen (Abschnitt 1.1.2). Für eine ausführliche Einführung verweisen wir auf den Grundlagenteil ii.

1.1.1 Zertifizierende Algorithmen

Laufzeitverifikation beschäftigt sich mit der Frage, ob ein Eingabe-Ausgabe-Paar korrekt ist und das zur Laufzeit. *Zertifizierende Algorithmen* bieten eine Möglichkeit der Laufzeitverifikation. Dabei überzeugt ein zertifizierender Algorithmus seinen Nutzer durch ein Korrektheitsargument zur Laufzeit. Das ist zum Beispiel insbesondere für ausgelagerte Berechnungen interessant. Der Nutzer hat keinen Zugriff auf den Algorithmus und vertraut den Entwickler:innen nicht blind.

Dafür berechnet ein zertifizierender Algorithmus für eine Eingabe zusätzlich zur Ausgabe noch einen *Zeugen* – ein leicht-zu-prüfender Beweis für die Korrektheit des Eingabe-Ausgabe-Paars. Als Beispiel betrachten wir einen Algorithmus, der entscheidet, ob ein ungerichteter Graph bipartit ist – also ob sich seine Knoten in zwei Partitionen A und B aufteilen lassen, sodass jede Kante einen Knoten aus A mit einem Knoten aus B verbindet. Nehmen wir an, der Algorithmus entscheidet für einen Graphen G, dass G nicht bipartit ist. Eine zertifizierende Variante berechnet zusätzlich einen ungeraden Kreis in G als Zeugen dafür, dass G nicht bipartit ist: ein ungerader Kreis ist bereits selbst nicht bipartit. Ein Nutzer kann sich davon selbst überzeugen, indem er die Knoten des ungeraden Kreises alternierend partitioniert und sieht, dass dies nicht gelingen kann.

Zu jedem zertifizierenden Algorithmus gehört ein *Zeugenprädikat* – ein Prädikat mit der folgenden Eigenschaft: wenn es mit einer Eingabe, einer Ausgabe und einem Zeugen erfüllt ist, dann folgt, dass das Eingabe-Ausgabe-Paar korrekt ist. Ein Zeugenprädikat, das erfüllt ist, wenn ein ungerader Kreis in einem Graphen enthalten ist, ist leicht zu prüfen, wenn der ungerade Kreis in G bereits als Zeuge gegeben ist.

Ein Nutzer muss das Zeugenprädikat nicht selbst entscheiden. Ein *Checker* ist ein Algorithmus, der das Zeugenprädikat für den Nutzer entscheidet. Die Korrektheit eines Checkers ist folglich zwingend notwendig für die Laufzeitverifikation mit einem zertifizierenden Algorithmus. Dadurch wird die *formale Instanzverifikation* motiviert, bei der wir Checker verifizieren und einen maschinen-geprüften Beweis für die Korrektheit eines Eingabe-Ausgabe-Paars zur Laufzeit gewinnen.

Zertifizierende *sequentielle* Algorithmen sind gut untersucht. Es gibt eine Theorie, über 100 Fallstudien, Entwurfsmuster, sowie ein Framework zur formalen Instanzverifikation für den Beweisassistenten Isabelle [McC+11; Riz15].

1.1.2 Verteilte Algorithmen

Verteilte Algorithmen sind so entworfen, dass sie auf einem verteilten System laufen. Ein *verteiltes System* besteht aus Komponenten, die lokal rechnen können und miteinander kommunizieren können. Bei einem verteilten Algorithmus lösen die Komponenten eines verteilten Systems gemeinsam *ein* Problem. Dafür beschreibt ein verteilter Algorithmus, wie die Berechnung und Kommunikation einer jeden Komponente aussieht. Während einige verteilte Algorithmen terminieren, laufen andere fortwährend.

1.2 FORSCHUNGSFRAGE

Die Forschungsfrage, mit der wir uns in der vorliegenden Arbeit auseinander setzen, ist die folgende:

> Wie können wir das Konzept zertifizierender *sequentieller* Algorithmen so auf *verteilte* Algorithmen übertragen, dass wir einerseits nah an dem ursprünglichen Konzept bleiben und andererseits die Gegebenheiten verteilter Systeme berücksichtigen?

In Abschnitt 1.2.1 motivieren wir zunächst, warum eine solche Übertragung interessant ist. Insbesondere diskutieren wir hierbei die beiden formulierten Ziele für eine Übertragung des Konzepts. In Ab-

schnitt 1.2.2 stellen wir Herausforderungen bei einer Übertragung des Konzepts heraus, aus denen sich detailliertere Unterfragen zur übergeordneten Forschungsfrage ergeben.

1.2.1 Motivation für eine Übertragung

Ein verteilter Algorithmus gibt eine Vorschrift für das Verhalten einer jeden Komponente eines verteilten Systems vor. Das Verhalten des verteilten Systems wiederum ergibt sich aus dem Zusammenspiel der Komponenten. Dieses Verhalten zu überschauen ist schwierig. Bereits in [LSP82] wird darauf verwiesen, dass entworfene verteilte Algorithmen oder deren Korrektheitsbeweise immer wieder fehlerhaft sind. In [Pel00] schreibt Peleg dazu:

> Discussing and analyzing the properties of distributed algorithms, and proving their correctness, also turns out to be considerably more challenging and problematic in the distributed setting than in the sequential one, and methods for formally reasoning about the behavior of distributed systems are among today's most active research areas in the field.

Auch Raynal beschreibt in [Ray13] diese Schwierigkeit:

> Although distributed algorithms are often made up of a few lines, their behavior can be difficult to understand and their properties hard to state and prove. Hence, distributed computing is not only a fundamental topic but also a challenging topic where simplicity, elegance, and beauty are first-class citizens.

1.2.1.1 *Aktueller Stand der Schwierigkeit*

Peleg formulierte seine Beobachtung zur Schwierigkeit verteilter Systeme circa 15 Jahre bevor die Autorin der vorliegenden Arbeit mit ihren Arbeiten zur formulierten Forschungsfrage begann. Die anschließende Forschung bis heute zeigt jedoch, dass diese Schwierigkeit weiterhin besteht [KNR12; Aba+13; Joh+13; NSH14; WB15; Wil+15a; BKZ15; ACD16; LSL17; ABG18; Ami+18; Ber+19].

Der Einsatz zertifizierender sequentieller Algorithmen ist eine Methode zur Laufzeitverifikation. Obwohl es sich bei der Laufzeitverifikation um ein jüngeres Forschungsgebiet handelt, gibt es bereits eine Vielzahl an Methoden für die Laufzeitverifikation verteilter Systeme [BKZ15; Hal16; DN17; FPS18; Gie+19]. Die meisten dieser Methoden beschreiben sequentielle Ansätze mit einem zentralen Monitor, der die Einhaltung einer Eigenschaft überwacht. Uns ist keine Methode zur Lauf-

zeitverifikation bekannt, die den Einsatz zertifizierender Algorithmen für verteilte Systeme beschreibt.

Wir sind dadurch motiviert zu untersuchen, wie zertifizierende verteilte Algorithmen aussehen können und ob das Konzept für die Laufzeitverifikation verteilter Systeme taugt.

1.2.1.2 *Zwei Ziele für eine Übertragung des Konzepts*

Wir verfolgen zwei teilweise gegensätzliche Ziele für eine Übertragung des Konzepts zertifizierender Algorithmen auf verteilte Algorithmen. Zum einen möchten wir nah am ursprünglichen Konzept der sequentiellen Zertifizierung bleiben und zum anderen möchten wir die Gegebenheiten verteilter Systeme berücksichtigen.

Wir möchten möglichst nah am ursprünglichen Konzept zertifizierender sequentieller Algorithmen bleiben, um zu untersuchen, ob und wie eine Laufzeitverifikation mit Zeugen, Zeugenprädikat und Checker in einem verteilten System gelingt. Im besten Fall ist ein zertifizierender verteilter Algorithmus dabei ein Spezialfall eines zertifizierenden sequentiellen Algorithmus, sodass wir bereits vorhandene Ergebnisse übertragen können. Darüber hinaus möchten wir die softwaretechnische Perspektive der sequentiellen Zertifizierung beibehalten. Wir untersuchen deswegen auch Kriterien zur Güte einer zertifizierenden Variante, Entwurfsmuster, sowie ein Framework zur formalen Instanzverifikation.

Eine direkte Übertragung des Konzepts auf verteilte Algorithmen sähe so aus, dass ein sequentieller Checker zentral ein Zeugenprädikat für ein verteiltes System entscheidet. Wir zeigen, dass diese direkte Übertragung jedoch nicht zu den Gegebenheiten eines verteilten Systems passt.

Wir möchten das Konzept deswegen nicht nur direkt, sondern auch so übertragen, dass wir dabei die Gegebenheiten verteilter Systeme berücksichtigen. Durch die wenigen Arbeiten auf dem Gebiet der verteilten Laufzeitverifikation sind wir zusätzlich motiviert, eine verteilte Laufzeitverifikation mit zertifizierenden verteilten Algorithmen zu erreichen. Hierbei sollen Zeugen verteilt berechnen werden und ein Zeugenprädikat durch einen verteilten Checker entschieden werden. Mit Fallstudien möchten wir außerdem untersuchen, ob eine solche Zertifizierung für verteilte Systeme interessant ist. Darüber hinaus soll auch ein Framework zur formalen Verifikation die Gegebenheiten eines verteilten Systems berücksichtigen. Verifizierte Checker sollen deswegen auf einem realen verteilten System laufen können.

1.2.2 Herausforderungen

Verteilte Algorithmen bergen spezielle Herausforderungen [Peloo], die wir in diesem Abschnitt im Kontext der Zertifizierung betrachten.

1.2.2.1 *Verteiltes Eingabe-Ausgabe-Paar*

Verteilte Algorithmen, die terminieren, berechnen eine *verteilte* Ausgabe für eine *verteilte* Eingabe; Dolev beschreibt diese Klasse verteilter Algorithmen in [Doloo] wie folgt:

> *A large class of distributed algorithms compute a fixed distributed output based on a distributed input. Usually the distributed input of a node is related to its local topology. [...] The distributed output is a function of the distributed input; the output usually marks links of the communication graph.*

Durch ein *verteiltes Eingabe-Ausgabe-Paar* entstehen einige Fragen an die Zertifizierung verteilter Algorithmen:

- Sollte ein Zeuge die Korrektheit eines verteilten Eingabe-Ausgabe-Paars, eines Eingabe-Ausgabe-Paars je Komponente oder alle Eingabe-Ausgabe-Paare in einem Teilnetzwerk verifizieren?
- Wie viele Zeugen sollte es geben? Beteiligen sich alle Komponenten bei der Berechnung eines Zeugen oder auch der Zeugen?
- Wie viele Checker sollte es für ein Netzwerk geben?

1.2.2.2 *Nicht-Terminierung*

Einige verteilte Algorithmen sind so entworfen, dass sie fortwährend laufen, wie zum Beispiel Kommunikationsprotokolle. Auch eine *Nicht-Terminierung* wirft Fragen für die Zertifizierung auf:

- Was soll zur Laufzeit verifiziert werden, wenn es kein Eingabe-Ausgabe-Paar gibt?
- Wann sollte ein nicht terminierender Algorithmus einen Zeugen berechnen?
- Wann sollte der Checker eines nicht terminierenden Algorithmus ein Zeugenprädikat prüfen?

In [AR05] argumentieren Arkoudas und Rinard, dass Zertifizierung für reaktive Systeme (also nicht terminierend) nicht interessant sei:

> *For other components, such as reactive systems, the important issue is not output correctness but behavioral safety [...]*

Dabei beziehen sie sich auf Datenstrukturen, die im Kontext sequentieller Algorithmen zwar vorkommen, aber eben nicht auf Terminierung ausgelegt sind. Bei der Forschung zur Zertifizierung stehen „klassi-

sche“ sequentielle Algorithmen, die terminieren, im Vordergrund. Dennoch gibt es auch vielversprechende Arbeiten zu zertifizierenden Datenstrukturen [McC+11; FM99; SM91; BS95].

1.2.2.3 *Kommunikation*

Bei einem verteilten Algortihmus muss die *Kommunikation* der Komponenten geregelt werden. Hierdurch ergeben sich die folgenden Fragen an die Zertifizierung:

- Wie sollte ein Checker in ein verteiltes System integriert werden? Ist ein Checker eine weitere Komponente in einem verteilten System?
- Wie und mit wem kommuniziert ein Checker?

1.2.2.4 *Unvollständiges Wissen*

In einem verteilten System kennt jede Komponente meist nur einen Teil des verteilten Systems, sowie einen Teil der nötigen Informationen zur Lösung eines Problems im System. Durch dieses *unvollständige Wissen* der Komponenten werden die folgenden Fragen für die Zertifizierung aufgeworfen:

- Was bedeutet unvollständiges Wissen für einen verteilten Zeugen? Sollte jede Komponente nur einen Teil eines Zeugen kennen oder jede den ganzen Zeugen?
- Was bedeutet unvollständiges Wissen für das Problem der Entscheidung eines Zeugenprädikats in einem System? Welche Informationen sollte eine Komponente dafür besitzen?

1.2.2.5 *Fehler*

Ein zertifizierender sequentieller Algorithmus schützt vor einem fehlerhaften Algorithmus, einer fehlerhaften Implementierung und einer fehlerhaften Ausführung. In einem verteilten System kann es zu Fehlern kommen, die keine Entsprechung im Sequentiellen finden. Zum Beispiel könnten nur einige Komponenten des Systems eine inkorrekte Ausgabe berechnen, während die anderen Komponenten korrekt rechnen. Durch die Betrachtung von *Fehlern* ergeben sich die folgenden Fragen für die Zertifizierung:

- Wie übertragen wir das Fehlermodell, das wir für die Zertifizierung eines sequentiellen Algorithmus annehmen, auf verteilte Algorithmen?
- Wie gehen wir mit dem Begriff der Korrektheit um, wenn eine verteilte Ausgabe teilweise korrekt sein kann?

2 ERGEBNISSE UND AUFBAU DIESER ARBEIT

Wir stellen in diesem Kapitel die Ergebnisse der vorliegenden Arbeit vor. Dabei folgen wir nicht exakt dem Aufbau der Arbeit. In Abschnitt 2.1 fassen wir die Ergebnisse kurz zusammen. In Abschnitt 2.2 erläutern wir den Aufbau der Arbeit, wobei wir zum Beispiel auch unseren Umgang mit eingebundenem Quellcode erläutern.

2.1 ERGEBNISSE DIESER ARBEIT

Wir stellen in Abschnitt 2.1.1 die Ergebnisse der vorliegenden Arbeit vor. In Abschnitt 2.1.2 erläutern wir die Veröffentlichungen, die im Rahmen dieser Arbeit entstanden sind.

2.1.1 Überblick über die Ergebnisse

2.1.1.1 *Zertifizierende terminierende verteilte Algorithmen*

Wir haben eine Methode entwickelt, das Konzept *zertifizierender sequentieller Algorithmen* auf *verteilte Algorithmen* zu übertragen. Dabei haben wir zwei Ziele berücksichtigt. Zum einen das Konzept so zu übertragen, dass es nah am ursprünglichen Konzept für sequentielle Algorithmen ist: Zeuge, Zeugenprädikat und Checker haben vergleichbare Rollen wie bei zertifizierenden sequentiellen Algorithmen. Zum anderen das Konzept aber auch so zu übertragen, dass es zu den Bedingungen eines verteilten Systems – bei uns meist Netzwerke – passt: zum Beispiel, dass ein Zeuge verteilt berechnet und geprüft wird.

Die beiden Ziele mussten wir dafür teilweise gegeneinander abwägen. Als Resultat dieser Abwägung haben wir uns für eine Klasse zertifizierender verteilter Algorithmen entschieden,

- die terminieren,
- ihr *verteiltes* Eingabe-Ausgabe-Paar verifizieren,
- einen *verteilten* Zeugen mit jeder Ausgabe berechnen und
- ein *verteilbares* Zeugenprädikat besitzen,
- das durch einen *verteilten* Checker entschieden wird.

Die von uns so gewählten zertifizierenden verteilten Algorithmen sind ein Spezialfall zertifizierender sequentieller Algorithmen.

Verteilte Zeugen & Konsistenz. Durch die Verteiltheit eines Zeugen ergeben sich einige Unterschiede zwischen Zeugen zertifizierender verteilter Algorithmen und Zeugen zertifizierender sequentieller Algorithmen. Wir haben deswegen Kriterien für verteilte Zeugen eingeführt, durch die wir grundlegende Ideen verteilter Algorithmen auch bei der Zertifizierung in der Wahl eines Zeugen widerspiegeln.

Für verteilte Zeugen haben wir außerdem das Problem der *Konsistenz* formuliert, für das sich bei zertifizierenden sequentiellen Algorithmen keine Entsprechung findet. Wenn ein Zeuge verteilt in einem Netzwerk vorliegt, dann besitzen einige Teile des Zeugen häufig einige geteilte Informationen. Damit der Zeuge ein schlüssiges Korrektheitsargument bildet, dürfen sich seine Teile in den geteilten Informationen nicht widersprechen.

Verteilbare Zeugenprädikate. Wir haben aufgezeigt, dass sequentielle Checker nicht zu den Bedingungen eines Netzwerks passen. Durch das Ziel verteilter Checker motiviert, haben wir das Konzept verteilbarer Zeugenprädikate eingeführt. Ein verteilbares Prädikat ist ein globales Prädikat in einem Netzwerk, das entschieden werden kann, indem lokale Prädikate für die Komponenten entschieden werden. Ein lokales Prädikat kann dabei für eine Komponente unabhängig von den restlichen Komponenten des Netzwerks entschieden werden.

2.1.1.2 *Entwurfsmuster*

Wir haben zertifizierende verteilte Algorithmen aus einem softwaretechnischen Blickwinkel betrachtet, wie es auch für zertifizierende sequentielle Algorithmen üblich ist. Hierfür haben wir die bekannten Entwurfsmuster zertifizierender sequentieller Algorithmen auf die Entwicklung zertifizierender verteilter Algorithmen übertragen.

Des Weiteren haben wir auch spezielle Entwurfsmuster für die Entwicklung zertifizierender verteilter Algorithmen herausgearbeitet. Darüber hinaus haben wir analysiert, mit welchen weiteren formalen Methoden der Einsatz zertifizierender verteilter Algorithmen geschickt kombiniert werden kann. Wir haben außerdem die Simulationsumgebung für zertifizierende verteilte Algorithmen bereitgestellt.

2.1.1.3 *Verteilte Checker*

Wir haben eine verteilte Architektur zur Integration von Checkern in ein Netzwerk beschrieben.

Entscheidung eines verteilbaren Zeugenprädikats. Ein Checker entscheidet ein verteilbares Zeugenprädikat verteilt. Dabei wird das ver-

teilbare Zeugenprädikat durch lokale Prädikate entschieden. Die Kommunikation im Netzwerk für die Kombination der lokalen Prädikate läuft dabei unabhängig von dem Zeugenprädikat nach immer dem gleichen Muster ab.

Konsistenzprüfung. Eine zusätzliche Aufgabe eines verteilten Checkers ist es außerdem die Konsistenz eines Zeugen zu prüfen. Wir haben eine Konsistenzprüfung für beliebige Zeugen entwickelt. Für diese ist eine Kommunikation im gesamten Netzwerk nötig. Wir haben gezeigt, dass wir beliebige Zeugen immer so erweitern können, dass Konsistenz auch lokal prüfbar ist. Dabei muss jede Komponente nur eine Nachricht an jeden Nachbarn senden.

2.1.1.4 *Industrielle Fallstudie: Verteilte Auktion*

Wir haben eine industrielle Fallstudie mit dem Industriepartner In-Systems durchgeführt: eine zertifizierende verteilte Auktion für Transportroboter in einer Fabrik. Die genannten Herausforderungen waren dabei eine unbekannte Umgebung für die Roboter und eine dezentrale Steuerung der Roboterflotte. Wir haben anhand der Fallstudie gezeigt, dass zertifizierende verteilte Algorithmen diesen Herausforderungen gerecht werden können.

Darüber hinaus haben wir mit der Fallstudie auch ein Multi-Agenten-System, anstelle eines Netzwerks untersucht und dabei virtuelle Netzwerke zur Reduktion der Invasivität (Grade des Ressourcenverbrauchs durch die Laufzeitverifikation) des Checkers eingesetzt.

2.1.1.5 *Lokale Korrektheit*

Wir haben das Problem der lokalen Korrektheit formuliert. Intuitiv gesprochen ist mit lokaler Korrektheit gemeint, dass eine Komponente des Netzwerks eine korrekte Ausgabe berechnet hat. Die Motivation für eine Untersuchung der lokalen Korrektheit ist, dass Laufzeitverifikation keine vollständige Korrektheit garantiert und somit zur Laufzeit fehlschlagen kann. In einem verteilten System schlägt sie jedoch auch fehl, wenn nur eine einzige Komponente fehlerhaft ist.

Anhand einer Fallstudie haben wir deswegen für diesen Fall untersucht, wie die Zertifizierung der gesamten Ausgabe des Netzwerks genutzt werden kann, um die lokale Korrektheit für einige Ausgaben der Komponenten zu verifizieren.

2.1.1.6 *Nicht-Terminierung*

Wir haben an einer Fallstudie eines Mutex-Algorithmus analysiert, wie Zertifizierung für verteilte Algorithmen, die nicht terminieren, gelingen kann. Die Übertragung des Konzepts orientiert sich dabei

weniger an zertifizierenden sequentiellen Algorithmen und mehr an zertifizierenden Datenstrukturen, die bisher im Vergleich noch wenig untersucht sind.

2.1.1.7 *Formale Instanzverifikation*

Methodik. Wir haben eine Methodik zur formalen Instanzverifikation für zertifizierende verteilte Algorithmen entwickelt. Dafür haben wir die nötigen Beweisverpflichtungen herausgestellt und eine Auswahl an Werkzeugen für die Entwicklung maschinen-geprüfter Beweise getroffen.

Framework in COQ. Wir haben ein Framework für die formale Instanzverifikation in COQ entwickelt, sodass ein verifizierter Checker für ein reales Netzwerk gewonnen werden kann. Wir haben das Framework an drei Fallstudien demonstriert. Des Weiteren haben wir die lokale und die allgemeine Konsistenzprüfung in COQ für das Framework implementiert.

2.1.2 Veröffentlichungen

Im Rahmen dieser Arbeit sind sechs Konferenzartikel entstanden und vier studentische Abschlussarbeiten, die von der Autorin dieser Arbeit betreut wurden. Alle Implementierungen, die Rahmen dieser Arbeiten in COQ entstanden sind, sind online bei dem Dienstleister GitHub zur Verwaltung quelloffener Software einsehbar. [1]

2.1.2.1 *Konferenzartikel*

Wir stellen die Konferenzartikel in Kürze vor und erläutern, welchen Anteil die Autorin dieser Arbeit jeweils daran hatte, falls es mehrere Autor:innen gibt.

Certification of Distributed Algorithms Solving Problems with Optimal Substructure – Völlinger & Reisig [VR15]. Die Autorin der vorliegenden Arbeit stellt als erste Fallstudie eine zertifizierende verteilte Variante zur Lösung des Problems der Berechnung kürzester Pfade in einem Netzwerk vor und diskutiert die Rolle der optimalen Substruktur des Problems. Reisig trug maßgeblich zu Form und Gestaltung des Artikels bei.

Verifying a Class of Certifying Distributed Programs – Völlinger & Akili [VA17]. Die Autorin der vorliegenden Arbeit stellt eine Methode der formalen Instanzverifikation vor und darüber hinaus eine zertifizierende Variante zur Lösung des Problems der Leader-Election in Netzwerken. Akili lieferte die Implementierung in COQ zur Fallstudie

1 https://github.com/voellinger/verified-certifying-distributed-algorithms

der zertifizierenden Leader-Election mit der vorgeschlagenen Methode zur formalen Instanzverifikation.

Verifying the Output of a Distributed Algorithm Using Certification – Völlinger [Völ17]. In dem Artikel wird das Konzept verteilbarer Prädikate zur Formulierung globaler Eigenschaften in Netzwerken eingeführt, sodass sie lokal entschieden werden können. Illustriert wird ein verteilbares Zeugenprädikat an der Fallstudie eines zertifizierenden verteilten Bipartitheitstests.

On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency – Völlinger & Akili [VA18]. Die Autorin der vorliegenden Arbeit führt das Konzept der Konsistenz ein, das für verteilte Zeugen nötig ist. Sie zeigt außerdem, dass eine lokale Entscheidung der Konsistenz für Zeugen möglich ist, wenn diese eine spezielle Form einhalten. Darüber hinaus ordnet sie die Entscheidung der Konsistenz in ein Framework zur formalen Instanzverifikation für den Beweisassistenten COQ ein. Akili liefert die Implementierung der lokalen Konsistenzprüfung in COQ.

Case Study on Certifying Distributed Algorithms: Reducing Intrusiveness – Akili & Völlinger [AV19]. In dem Artikel beschreiben Akili und Völlinger eine gemeinsam durchgeführte industrielle Fallstudie zur Zertifizierung einer verteilten Auktion in einem Multi-Agenten-System. Sie reduzieren die Invasivität des Checkers, also den Grad zu dem der Checker durch Ressourcenverbrauch in das ursprüngliche System eingreift, durch die Einführung von virtuellen Netzwerken für die Kommunikation des Checkers.

On Certifying Distributed Algorithms: Problem of Local Correctness – Völlinger [Völ19]. In dem Artikel wird das Problem der lokalen Korrektheit eingeführt und anhand einer zertifizierenden verteilten Variante der Berechnung kürzester Pfade illustriert.

Darüber hinaus möchten wir eine weitere Veröffentlichung [AL19] erwähnen, bei der die Autorin dieser Arbeit allerdings nicht mitgewirkt hat. Akili baut auf der gemeinsamen industriellen Fallstudie [AV19] auf; gemeinsam mit Lorenz zeigt sie eine Kombination der zertifizierenden verteilten Auktion mit zwei Techniken der Laufzeitverifikation auf, sodass neben funktionalen Eigenschaften auch Eigenschaften darüber verifiziert werden, wie die Kommunikation abläuft und das Deadlines eingehalten.

2.1.2.2 *Studentische Abschlussarbeiten*

Wir stellen in Kürze die studentischen Abschlussarbeiten vor, die im Kontext dieser Arbeit unter der Betreuung der Autorin entstanden sind.

Simulation Zertifizierender Netzwerkalgorithmen – Akili. Akili befasst sich in ihrer Bachelorarbeit [Aki15] mit der Simulation zertifizierender verteilter Algorithmen. Sie illustriert ihre Simulationsumgebung mit einem zertifizierenden verteilten Bellman-Ford-Algorithmus zur Berechnung kürzester Pfade in einem Netzwerk.

Verifikation eines zertifizierenden verteilten Algorithmus – Asher. Asher beweist in seiner Diplomarbeit [Ash16] die Zeugeneigenschaft und die Verteilungseigenschaft eines Zeugenprädikats für das Problem der Berechnung der kürzesten Pfade in einem Netzwerk in COQ.

Verteilte Prüfung der Konsistenz im Rahmen eines Verifikations-Frameworks für Zertifizierende Verteilte Algorithmen – Akili. Akili implementiert in ihrer Masterarbeit [Aki18] eine lokale Konsistenzprüfung für zusammenhängende Zeugen in COQ und integriert diese in das Framework zur formalen Instanzverifikation.

Formale Instanzverifikation zertifizierender Algorithmen: Fallstudie Zweifärbbarkeit & alternative Konsistenzprüfung – Boll. Boll implementiert in seiner Diplomarbeit [Bol19] eine Konsistenzprüfung für beliebige Zeugen und integriert sie in das Framework zur formalen Instanzverifikation in COQ. Darüber hinaus setzt er auch die formale Instanzverifikation für den zertifizierenden verteilten Bipartitheitstest in dem Framework um.

2.2 AUFBAU DIESER ARBEIT

Wir stellen in diesem Abschnitt den Aufbau der vorliegenden Arbeit vor. Dafür geben wir in Abschnitt 2.2.1 zunächst einen Überblick über alle Teile dieser Arbeit in entsprechender Reihenfolge. In Abschnitt 2.2.2 begründen wir die Auswahl der verteilten Algorithmen für unsere Fallstudien. In Abschnitt 2.2.3 erläutern wir dann unseren Umgang mit Quellcode in dieser Arbeit und in Abschnitt 2.2.4 unseren Umgang mit inklusiver Sprache.

2.2.1 Übersicht über die Teile dieser Arbeit

Grundlagen. In Teil ii legen wir die Grundlagen für diese Arbeit. Wir führen zertifizierende sequentielle Algorithmen und verteilte Algorithmen ein.

Zertifizierende terminierende verteilte Algorithmen. In Teil iii stellen wir unsere Methode zur Übertragung des Konzepts zertifizierender sequentieller Algorithmen auf verteilte Algorithmen vor. Wir führen verteilbare Zeugenprädikate ein und beschäftigen uns eingehend mit verteilten Zeugen, für die wir das Problem der Konsistenz formulieren.

Verteilte Checker. In Teil iv widmen wir uns verteilten Checkern. Wir stellen die verteilte Prüfung verteilbarer Zeugenprädikate vor, sowie die dafür nötige Konsistenzprüfung verteilter Zeugen.

Formale Instanzverifikation. In Teil v führen wir das Framework für die formale Instanzverifikation in COQ ein und stellen die entsprechenden Fallstudien vor.

Entwurfsmuster & weitere Klassen. In Teil vi diskutieren wir Entwurfsmuster zur Entwicklung zertifizierender Varianten verteilter Algorithmen. Des Weiteren stellen wir mit der Fallstudie zur lokalen Korrektheit und der Fallstudie zur Nicht-Terminierung auch weitere Klassen zertifizierender verteilter Algorithmen vor.

Diskussion. In Teil vii fassen wir die Ergebnisse der vorliegenden Arbeit zusammen und diskutieren welche Forschungsfragen sich anschließen könnten. Hierbei betrachten wir vor allem weitere Methoden der Übertragung des Konzepts zertifizierender sequentieller Algorithmus auf verteilte Algorithmen.

Anhang: Mathematische Notationen. Wir zeigen auf welche Notationen wir in welchem Kontext benutzen. Wir führen dabei jedoch nicht die zugehörigen mathematischen Grundlagen ein, sondern setzen eine entsprechende mathematische Grundbildung unserer Leser:innen voraus.

2.2.2 Auswahl der Fallstudien

Zunächst einmal möchten wir hervorheben, dass wir uns in dieser Arbeit mit verteilten Algorithmen und nicht mit verteiltem Rechnen beschäftigen. Verteilte Algorithmen bilden die Grundlage für das verteilte Rechnen, indem sie Lösungen für grundlegende Probleme bieten, die durch eine Verteilung einer Berechnung entstehen. Dabei bieten sie zum Beispiel Lösungen für die Kommunikation in einem verteilten System, zur Koordination der Komponenten, zum Zugriff auf Ressourcen, zum Umgang mit Fehlern, zur Suche auf den Komponenten, zum Symmetrie brechen zwischen Komponenten oder auch zur Synchronisation der Komponenten.

Die Auswahl der Fallstudien der vorliegenden Arbeit sind inspiriert durch Referenzwerke zu verteilten Algorithmen [Lyn96; Gho14; Ray13; Tel94; Peloo; AW04; Erc13]. Eine detailliere Begründung zur konkreten Auswahl geben wir, wenn wir die jeweilige Fallstudie vorstellen.

2.2.3 Quellcode

In Teil v binden wir Quellcode ein, dabei handelt es sich um Ausschnitte größerer Implementierungen. Die gesamte Implementierung ist dabei jeweils auf GitHub einsehbar. [2]

2.2.3.1 *Abweichende Benennung*

Alle Implementierungen sind auch im Rahmen verschiedener Arbeiten veröffentlicht worden. Da sich über die Zeit jedoch Benennungen verändert haben, sind die Benennungen in dieser Arbeit gegebenenfalls nicht immer konsistent mit der Benennung auf GitHub oder den Benennungen in den entsprechenden Arbeiten, in denen die Implementierung veröffentlicht wurde. Wir haben Benennungen so angepasst, dass sie zum Kontext dieser Arbeit passen.

2.2.3.2 *Vereinfachungen*

Darüber hinaus zeigen wir nicht jedes Detail der Implementierungen und erklären auch nicht jedes Detail der Ausschnitte einer Implementierung, die wir zeigen. Wir beschränken uns auf ausgewählte Ausschnitte, die dazu dienen den Leser:innen einen Eindruck zu vermitteln. Wir verweisen jedoch auch immer auf die Arbeit, in der die jeweilige Implementierung veröffentlicht ist. Interessierte Leser:innen haben damit die Möglichkeit sich vertieft mit der Implementierung auseinanderzusetzen.

Um das Verständnis zu erhöhen, vereinfachen wir auch Teile des Quellcodes. An dieser Stelle ist uns der Lesefluss wichtiger als Exaktheit und Vollständigkeit.

2.2.4 Inklusive Sprache

Wir bemühen uns in dieser Arbeit um eine inklusive Sprache. Wir verwenden deswegen durchgängig ein inklusives Plural für alle Menschen. Zum Beispiel schreiben wir: *Leser:innen*. Unsere Hoffnung ist, dass sich damit alle Menschen gleichermaßen angesprochen fühlen.

Neben dem Plural gibt es zwei durchgängige Rollen in dieser Arbeit, über die wir im Singular schreiben. Hier verwenden wir einmal durchgängig eine männliche und einmal durchgängig eine weibliche Form. Wir sprechen in der gesamten Arbeit von einem Nutzer und von einer Entwicklerin eines Algorithmus. Selbstverständlich ist die Geschlechterzuordnung dieser Rollen vertauschbar.

2 https://github.com/voellinger/verified-certifying-distributed-algorithms

Alle Menschen, die nicht-binär verortbar sind, sollen sich dennoch angesprochen fühlen [Ain15]. Wir entscheiden uns für diese Variante zum einen, um unseren Leser:innen einen guten Lesefluss zu gewähren und zum anderen aus Mangel an uns bekannten eleganten Alternativen in der deutschen Sprache.

Teil II

GRUNDLAGEN

Wir legen in diesem Teil die Grundlagen für die vorliegende Arbeit. In Kapitel 3 führen wir zertifizierende sequentielle Algorithmen ein. Wir erläutern dabei auch die Methode der formalen Instanzverifikation basierend auf zertifizierenden sequentiellen Algorithmen.

In Kapitel 4 führen wir verteilte Systeme ein und vor allem verteilte Algorithmen. Darüber hinaus diskutieren wir den Stand der Technik zur Laufzeitverifikation.

3 ZERTIFIZIERENDE SEQUENTIELLE ALGORITHMEN

Wir führen in diesem Kapitel zertifizierende sequentielle Algorithmen ein, wie sie in [McC+11] beschrieben sind. Für eine Übersicht zum Stand der Technik im Allgemeinen verweisen wir auf Kapitel 4.

In Abschnitt 3.1 formulieren wir das Problem, das durch eine zertifizierende Variante eines Algorithmus gelöst wird: das Instanzverifikationsproblem. In Abschnitt 3.2 stellen wir die Konzepte vor, mit denen ein zertifizierender Algortihmus das Instanzverifikationsproblem zur Laufzeit löst. In Abschnitt 3.3 führen wir darauf aufbauend die formale Instanzverifikation ein.

3.1 INSTANZVERIFIKATIONSPROBLEM

Das *Instanzverifikationsproblem* ist in der Abbildung 1 formuliert. Eine Spezifikation ist dabei als eine Vorbedingung und Nachbedingung gegeben und sie spezifiziert somit ein Problem über den Mengen I und O. Da wir uns mit Problemen beschäftigen, die durch einen Algorithmus gelöst werden soll, sprechen wir von Eingaben I und Ausgaben O und definieren die Vorbedingung über Eingaben, sowie die Nachbedingung über Eingaben und Ausgaben. Dabei ist die Spezifikation eine übliche *Eingabe-Ausgabe-Spezifikation* und es soll entsprechend für alle Eingabe-Ausgabe-Paare gelten:

$$\forall i \in I, o \in O : (i, o) \in \psi \vee i \notin \phi.$$

Das Instanzverifikationsproblem stellt die Frage, ob ein konkretes Eingabe-Ausgabe-Paar das spezifizierte Problem löst. Mit einer *Instanz* ist in diesem Kontext eine Eingabe gemeint. Ein Algorithmus, dem wir nicht blind vertrauen, berechnet für eine Eingabe (Instanz) eine Ausgabe und wirft damit das Instanzverifikationsproblem auf.

Ein Problem könnte das Lösen quadratischer Gleichungen $ax^2 + bx + c = 0$ sein und eine Instanz dieses Problems das Lösen der quadratischen Gleichung $x^2 + 2x + 1 = 0$. Nehmen wir an, wir sind an einer Lösung für die Unbekannte x interessiert. Wir benutzen einen Algorithmus, der die Instanz, an der wir interessiert sind, als Eingabe erhält. Nehmen wir weiterhin an, als Ausgabe berechnet er dann für x den Wert -1. Da wir dem Algorithmus nicht blind vertrauen, möchten

Gegeben:
Eingaben I
Ausgaben O
Spezifikation $\phi \subseteq I$, $\psi \subseteq I \times O$
Eingabe und Ausgabe: $i \in I$ und $o \in O$

Instanzverifikationsproblem:
Frage: $(i, o) \in \psi$ oder $i \notin \phi$?

Abbildung 1: Instanzverifikationsproblem formuliert für Probleme, die durch sequentielle Algorithmen gelöst werden sollen.

wir uns davon überzeugen, dass das berechnete x tatsächlich eine Lösung ist.

Wir sind folglich an der Lösung des Instanzverifikationsproblems für das Eingabe-Ausgabe-Paar $(x^2 + 2x + 1 = 0, -1)$ interessiert. In diesem Fall, können wir das Instanzverifikationsproblem ganz leicht lösen. Wir wenden die aus der Schule bekannte „Probe“ an, indem wir jedes Vorkommen von x in der Gleichung durch -1 ersetzen, den linken Term ausrechnen und die Gleichheit überprüfen.

Im Allgemeinen ist es jedoch nicht leicht, das Instanzverifikationsproblem zu lösen. In den meisten Fällen ist es genauso schwierig, wie eine korrekte Ausgabe für eine Eingabe zu berechnen.

3.2 ZEUGENPRÄDIKAT UND CHECKER

Die Idee eines zertifizierenden Algorithmus ist es, neben einer Ausgabe zusätzliche Informationen zu berechnen, mit denen sich ein Nutzer des Algorithmus von der Korrektheit der Ausgabe für seine Eingabe überzeugen kann – also das Instanzverifikationsproblem lösen kann. Diese Informationen liegen uns in einem mathematischen Objekt vor, das wir *Zeuge* nennen.

In Abschnitt 3.2.1 führen wir das Konzept eines Zeugen formal ein. Dafür benutzen wir *Zeugenprädikate*. In Abschnitt 3.2.2 führen wir dann *Checker* ein – Algorithmen, die einen Zeugen benutzen, um ein Zeugenprädikat zu entscheiden und darüber das Instanzverifikationsproblem für ein Eingabe-Ausgabe-Paar zu lösen.

3.2.1 Zeugenprädikate

Zu jedem zertifizierenden Algorithmus gehört ein *Zeugenprädikat* – ein Prädikat mit der folgenden Eigenschaft: wenn es mit einer Eingabe, einer Ausgabe und einem Zeugen erfüllt ist, dann folgt, dass

das Eingabe-Ausgabe-Paar korrekt ist. Wir definieren Zeugenprädikate und Zeugen: dabei sind die Mengen I, O und W intuitiv die Eingaben, Ausgaben und (potenziellen) Zeugen eines zertifizierenden Algorithmus.

Definition 1 (Zeugenprädikat, Zeugeneigenschaft, Zeuge)**.** *Seien* I, O, W *Mengen. Sei* (ϕ, ψ) *eine Spezifikation mit* $\phi \subseteq I$ *und* $\psi \subseteq I \times O$*. Sei* $\gamma \subseteq I \times O \times W$ *ein Prädikat mit der folgenden Eigenschaft:*

$$\forall i \in I, o \in O, w \in W : \gamma(i, o, w) \longrightarrow (\psi(i, o) \vee \neg\phi(i))$$

(i) γ *ist ein* Zeugenprädikat *für* (ϕ, ψ)*.*

(ii) Die Eigenschaft des Zeugenprädikats ist die Zeugeneigenschaft *für* (ϕ, ψ)*.*

(iii) $w \in W$ *ist ein* Zeuge *für die Korrektheit von* $(i, o) \in I \times O$ *bezüglich* (ϕ, ψ)*, falls* $(i, o, w) \in \gamma$*.*

Wir ordnen die Konzepte in den Kontext eines zertifizierenden Algorithmus ein und illustrieren sie dann gemeinsam an einem Beispiel. Ein *zertifizierender Algorithmus* für eine Spezifikation (ϕ, ψ) besitzt ein Zeugenprädikat γ für (ϕ, ψ). Darüber hinaus berechnet er für jede Eingabe i nicht nur eine Ausgabe o, sondern auch einen Zeugen w. Es gilt dann $(i, o, w) \in \gamma$ und somit die Korrektheit des Eingabe-Ausgabe-Paars (i, o) für die Spezifikation (ϕ, ψ).

Wenn wir ein Zeugenprädikat für ein Tripel positiv entscheiden, dann entscheiden wir das Instanzverifikationsproblem positiv für das entsprechende Eingabe-Ausgabe-Paar. Hier lässt sich das Potenzial des Einsatzes eine zertifizierenden Algorithmus erkennen: Wählen wir Zeugen und ein Zeugenprädikat geschickt, so ermöglichen wir eine leichtere Entscheidung des Instanzverifikationsproblems.

3.2.1.1 *Beispiel: Zertifizierender Bipartitheitstest*

Wir demonstrieren die Konzepte am Beispiel des Problems zu entscheiden, ob ein Graph bipartit ist. Ein Graph ist *bipartit*, falls sich seine Knoten so in zwei Klassen A und B partitionieren lassen, dass jede Kante einen Knoten in A mit einem Knoten in B verbindet.

Nehmen wir an, ein Algorithmus entscheidet für einen Graphen G, dass G nicht bipartit ist. Dann ist ein ungerader Kreis K, der als Teilgraph in G enthalten ist, ein *Zeuge* dafür, dass G nicht bipartit ist. Das liegt daran, dass ein ungerader Kreis bereits selbst nicht bipartit ist. Eine *zertifizierende* Variante des Algorithmus berechnet zusätzlich zur Ausgabe „G nicht bipartit“ noch K als Zeugen. Ein *Zeugenprädikat* für das Problem ist ein Prädikat das erfüllt ist, wenn ein ungerader Kreis in einem Graph enthalten ist. Es hat die *Zeugeneigenschaft* für das Problem, da ein ungerader Kreis bereits selbst nicht bipartit ist

und ein Graph mit einem nicht bipartiten Teilgraph insgesamt nicht bipartit ist.

Das Zeugenprädikat ist leicht für den Nutzer zu prüfen, wenn der ungerade Kreis K in G als Zeuge zusätzlich gegeben ist. Er kann sich auch von der Zeugeneigenschaft für seinen Graphen G überzeugen, indem er die Knoten von K alternierend partitioniert und sieht, dass dies nicht gelingen kann.

3.2.1.2 *Inkorrekter zertifizierender Algorithmus*

Wir sind bei der Beschreibung davon ausgegangen, dass der zertifizierende Algorithmus selbst korrekt ist. Das Ziel einer Entwicklerin eines zertifizierenden Algorithmus ist es, den Algorithmus so zu entwerfen, dass er für jede Eingabe sowohl eine korrekte Ausgabe als auch einen Zeugen berechnet. Wichtig ist aber zu sehen, dass der Nutzer dem Algorithmus nicht vertrauen muss, um sich von der Korrektheit seines Eingabe-Ausgabe-Paars zu überzeugen. Wenn der zertifizierende Algorithmus für seine Eingabe eine inkorrekte Ausgabe oder keinen Zeugen berechnet, dann schlägt die Verifikation zur Laufzeit fehl. Für den Nutzer ist das ersichtlich, da das Zeugenprädikat dann nicht erfüllt ist.

Darüber hinaus ist es für den Nutzer irrelevant, ob es immer einen Zeugen für jedes korrekte Eingabe-Ausgabe-Paar gibt. Die Zeugeneigenschaft fordert deswegen auch nur eine Implikation. Es ist die Aufgabe der Entwicklerin einer zertifizierenden Variante dafür zu sorgen, dass ihr zertifizierender Algorithmus korrekt ist. Ein Nutzer kann den zertifizierenden Algorithmus jedoch benutzen ohne sich von dessen Korrektheit zu überzeugen. Er überzeugt sich nur von der Korrektheit seines Eingabe-Ausgabe-Paars.

3.2.2 Checker

Wir sind bisher davon ausgegangen, dass der Nutzer eines zertifizierenden Algorithmus das Zeugenprädikat selbst entscheidet. Diese Aufgabe kann jedoch auch von einem Algorithmus übernommen werden. Ein Algorithmus, der das Zeugenprädikat für den Nutzer entscheidet, ist ein *Checker*. Wenn ein Checker also für ein Tripel aus Eingabe, Ausgabe und potenziellen Zeugen akzeptiert, dann ist das Eingabe-Ausgabe-Paar korrekt und der potenzielle Zeuge ist ein Zeuge.

Checker sind meist simpler aufgebaut als der zertifizierende Algorithmus. Das liegt daran, dass Zeugen und Zeugenprädikat so gewählt werden, dass sie die Instanzverifikation einfacher gestalten als die Berechnung einer korrekten Ausgabe für eine Eingabe.

Die Herausforderung bei der Entwicklung eines zertifizierenden Algorithmus ist es Zeugen und Zeugenprädikate zu finden, die diesem Anspruch gerecht werden. Für das Kriterium „einfacher“ werden in [McC+11] verschiedene Möglichkeiten diskutiert: zum Beispiel eine geringere Laufzeit, eine einfachere logische Struktur oder ein Checker, dessen Korrektheit bekannt ist im Gegensatz zur Korrektheit des zertifizierenden Algorithmus.

3.3 FORMALE INSTANZVERIFIKATION

Ein Nutzer eines zertifizierenden Algorithmus mit einem Checker muss noch ein gewisses Vertrauen aufbringen, um von der Korrektheit seines Eingabe-Ausgabe-Paars überzeugt zu sein. Er muss dafür sich sowohl von der Korrektheit der Zeugeneigenschaft als auch von der Korrektheit des Checkers überzeugen oder eben darauf vertrauen. In [McC+11] gibt es verschiedene Ansätze, die das Vertrauen thematisieren, welches einem Nutzer abverlangt wird. Einer dieser Ansätze ist die *formale Instanzverifikation*. Zu dieser hat vor allem Rizkallah im Rahmen ihrer Doktorarbeit geforscht [Alk+11; Alk+14; NRM14; Riz14; Riz15].

In Abschnitt 3.3.1 stellen wir die von Rizkallah entwickelte Methode, sowie das von ihr entworfene Framework in dem Beweisassistenten Isabelle zur Umsetzung ihrer Methode vor. Da wir in der vorliegenden Arbeit auch einen Beweisassistenten einsetzen, geben wir außerdem in Abschnitt 3.3.2 einen kurzen Überblick zu Beweisassistenten.

3.3.1 Methode und Framework

Die *formale Instanzverifikation* bedeutet, dass ein Nutzer einen maschinen-geprüften Beweis für die Korrektheit seines Eingabe-Ausgabe-Paars zur Laufzeit erhält, sofern die Laufzeitverifikation nicht fehlschlägt. Setzen wir die formale Instanzverifikation für einen zertifizierenden Algorithmus um, so liegt ein maschinen-geprüfter Beweis vor, sofern der Checker akzeptiert. Die Methode ist in Analogie zur formalen Verifikation benannt, bei der zur Designzeit ein maschinengeprüfter Beweis für die Korrektheit eines implementierten Algorithmus vorliegt und somit für die Korrektheit aller Eingabe-Ausgabe-Paare.

Rizkallah zeigt, dass für die formale Instanzverifikation eines zertifizierenden Algorithmus folgendes genügt:

- ein formal verifizierter implementierter Checker und

- ein maschinen-geprüfter Beweis für die Zeugeneigenschaft des Zeugenprädikats.

Sowie es für die formale Verifikation verschiedene Methoden der werkzeuggestützten Umsetzung gibt, gilt dies natürlich auch für die formale Instanzverifikation. Rizkallah setzt ihre Methode als Framework in dem Beweisassistenten Isabelle um. Dafür bindet sie teilweise noch weitere Werkzeuge ein. Sie stellt ein Framework in zwei Varianten vor: einmal wird die Verifikation der Checker in der Programmiersprache C einmal außerhalb von Isabelle und einmal innerhalb von Isabelle umgesetzt. Sie illustriert ihr Framework anhand von vier Fallstudien.

3.3.2 Beweisassistenten

Ein Beweisassistent ist ein Programm, das seinen Nutzer bei der Beweisführung unterstützt und die Korrektheit des geführten Beweises überprüft. Dafür stellt ein Beweisassistent implementierte Taktiken bereit, die Beweisschritte kleinschrittig ermöglichen, aber meist auch einige Automatisierungen bereitstellen.

Der geführte Beweis wird in einem Beweisskript festgehalten, welches in Abhängigkeit von dem Beweisassistenten mehr oder auch weniger verständlich für einen Menschen ist [Geu09]. Bekannte Beweisassistenten sind zum Beispiel Agda [Agd], Isabelle [Isa], NuPRL [Nup], HOL Light [Hol], LEGO [Leg], Mizar [Miz], MINLOG [Min] oder COQ [Coqa].

In der vorliegenden Arbeit setzen wir den Beweisassistenten COQ für ein Framework zur formalen Instanzverifikation für zertifizierende verteilte Algorithmen ein. COQ basiert auf dem Kalkül der induktiven Konstruktion – ein typisierter λ-Kalkül höherer Ordnung, der eine funktionale Programmiersprache und eine Logik höherer Ordnung kombiniert [PM15]. Die funktionale Programmiersprache von COQ erlaubt nur strukturelle Rekursion, wodurch die Terminierung von Funktionen stets garantiert ist. Infolgedessen ist die Sprache jedoch nicht Turing-mächtig.

Mit COQ können unter anderem mathematische Theorien und Spezifikationen formuliert werden, sowie ausführbare Funktionen definiert werden, die in einige übliche funktionale Programmiersprachen per Knopfdruck übersetzt werden können.

4 VERTEILTE SYSTEME

Ein verteilter Algorithmus wird so entworfen, dass er auf einem verteilten System ein Problem löst. In Abschnitt 4.1 führen wir verteilte Systeme im Allgemeinen und Netzwerke als eine Klasse verteilter Systeme im Speziellen ein. Anschließend führen wir in Abschnitt 4.2 verteilte Algorithmen ein.

In Abschnitt 4.3 diskutieren wir verwandte Arbeiten, indem wir den Stand der Technik zur Laufzeitverifikation betrachten.

4.1 NETZWERKE

Wir betrachten in Abschnitt 4.1.1 zunächst die Eigenschaften, die verteilte Systeme im Allgemeinen definieren und in Abschnitt 4.1.2 dann die Eigenschaften, die Netzwerke im Speziellen definieren.

4.1.1 Eigenschaften verteilter Systeme

Die Definitionen für ein *verteiltes System* sind in der Literatur nicht einheitlich, sie haben aber meist die folgenden Eigenschaften gemein:

- Ein verteiltes System besteht aus rechnenden Komponenten.
- Jede Komponente rechnet für sich, mehr oder weniger unabhängig von den anderen Komponenten des Systems.
- Die Komponenten kommunizieren miteinander.
- Die Komponenten lösen gemeinsam ein Problem.

Diese Eigenschaften eines verteilten Systems finden wir in vielen Referenzwerken zu verteilten Algorithmen wieder [Gho14; Ray13; Lyn96; Peloo; AW04]. Über diese Eigenschaften hinaus können wir zwischen verschiedenen Klassen verteilter Systeme unterscheiden.

Wir unterscheiden verteilte Systeme zum Beispiel nach ihrem Kommunikationsmedium. Die beiden gängigen Modelle sind hierbei das Modell des *Nachrichtenaustauschs* und das Modell des *geteilten Speichers* [Ray13; Peloo; Lyn96].

Die Komponenten bei dem Modell des Nachrichtenaustauschs kommunizieren über Nachrichtenkanäle, indem sie sich Nachrichten schicken. Bei dem Modell des geteilten Speichers kommunizieren die

Komponenten miteinander, indem sie im geteilten Speicher schreiben oder lesen.

4.1.2 Eigenschaften von Netzwerken

Wir betrachten in dieser Arbeit das Modell des Nachrichtenaustauschs und bezeichnen ein solches verteiltes System, wie in der Literatur üblich, als *Netzwerk*. Ein Netzwerk besteht also aus Komponenten und Nachrichtenkanälen.

4.1.2.1 *Komponenten*

Jede *Komponente* ist eine rechnende Einheit, die über einen eigenen Speicher verfügt und eigenständig rechnen kann. Eine Komponente kann zum Beispiel ein Prozess, ein Satellit, ein Smartphone oder auch ein PC sein.

Die verschiedenen Komponenten können dabei auch heterogen sein. Wie für die Betrachtung verteilter Algorithmen üblich, abstrahieren wir jedoch von der Heterogenität der Komponenten [Gho14; Ray13; Lyn96; Peloo; AW04].

4.1.2.2 *Nachrichtenkanäle*

Ein Nachrichtenkanal verbindet jeweils zwei Komponenten (engl. point-to-point communication) [Peloo]. Ein Kanal repräsentiert zum Beispiel ein Kabel, eine Verbindung über ein Internetprotokoll oder auch eine UNIX-Pipe. Wir betrachten Nachrichtenkanäle jedoch auf einer höheren Abstraktionsebene und interessieren uns deswegen nicht für die technische Umsetzung eines Kanals beziehungsweise der technischen Umsetzung der Kommunikation auf einem Kanal.

Wir bezeichnen zwei Komponenten, die über einen Nachrichtenkanal verbunden sind als *Nachbarn*. Alle Nachbarn einer Komponente sind zusammen die *Nachbarschaft* dieser Komponente.

Jede Komponente kann direkt mit ihren Nachbarn kommunizieren, indem sie eine Nachricht an einen Nachbarn sendet. Wir betrachten *bidirektionale* Kanäle. Das heißt, eine Nachricht kann von beiden Komponenten, die über den Kanal verbunden sind, sowohl verschickt als auch empfangen werden.

4.1.2.3 *Indirekte Kommunikation*

Ein Netzwerk ist immer *zusammenhängend*. Das heißt, zwischen zwei beliebigen Komponenten gibt es eine Verbindung über eine Folge von Komponenten und Kanälen. Daraus folgt, dass eine Komponente

über diese Verbindung mit allen nicht benachbarten Komponenten des Netzwerks kommunizieren kann.

Dafür sendet sie eine Nachricht an einen Nachbarn, der auf der Verbindung zum Empfänger der Nachricht liegt. Die Komponenten auf der Verbindung leiten die Nachricht entsprechend weiter (engl. end-to-end communication). Zur Umsetzung dieser Kommunikation bedarf es eines verteilten Algorithmus, wie zum Beispiel eines Routing-Protokolls [Med10].

4.1.2.4 *Synchrone und asynchrone Arbeitsweise*

Ein *synchrones* Netzwerk hat eine globale Uhr, durch die Komponenten in ihrer Arbeitsweise synchronisiert werden. Ein Netzwerk ohne globale Uhr ist ein *asynchrones* Netzwerk – jede Komponente arbeitet in ihrer eigenen Geschwindigkeit.

4.2 VERTEILTE ALGORITHMEN

Wir beschreiben in Abschnitt 4.2.1 zunächst welche Probleme verteilte Algorithmen lösen. In Abschnitt 4.2.2 führen wir verteilte Algorithmen ein, indem wir die Algorithmen der Komponenten und deren Zusammenspiel für einen verteilten Algorithmus beschreiben. In Abschnitt 4.2.3 diskutieren wir die Rolle der Terminierung für verteilte Algorithmen.

4.2.1 Probleme verteilter Algorithmen

Verteilte Algorithmen bilden die Grundlage für das verteilte Rechnen. Sie bieten Lösungen für die grundlegenden Probleme, die durch eine Verteilung der Berechnung entstehen.

Typische Probleme, die verteilte Algorithmen lösen, sind zum Beispiel die Kommunikation der Komponenten, die Koordination der Komponenten unter einander, den Zugriff auf geteilte Ressourcen, die Suche im System auf den Komponenten, das Brechen der Symmetrie zwischen Komponenten, die Synchronisation der Komponenten oder auch die Berechnung von Strukturen, wie einen Spannbaum in einem System.

Dabei lösen die Komponenten für einen verteilten Algorithmus gemeinsam *ein* Problem. Da wir von Netzwerken ausgehen und keinen verteilten Systemen, die Komponenten in verschiedene Gruppen einteilen, verhalten sich dabei alle Komponenten im Wesentlichen gleich.

Häufig werden verteilte Algorithmen zur Lösung eines gemeinsamen Problems mit sequentiellen Algorithmen für die Komponenten zum Lösen individueller Probleme verbunden. Dadurch gibt es dann auch eine Differenzierung im Verhalten der einzelnen Komponenten. Es können auch Teilnetzwerke gebildet werden, sodass in den Teilnetzwerken wiederum jeweils ein Problem mit einem verteilten Algorithmus gelöst wird.

4.2.2 Teilalgorithmen und verteilter Algorithmus

Bei einem verteilten Algorithmus führt jede Komponente eines Netzwerks Berechnungen für sich aus und kommuniziert mit ihren Nachbarn [Gho14; Ray13; Lyn96; Peloo; AW04]. Der Algorithmus einer Komponente beschreibt ihre Berechnungen und ihre Kommunikation. Im Kontext eines verteilten Algorithmus ist der Algorithmus einer Komponente der *Teilalgorithmus* dieser Komponente. Die Bezeichnungen in der Literatur für Teilalgorithmen sind nicht einheitlich.

Alle Komponenten besitzen für einen verteilten Algorithmus den gleichen Teilalgorithmus und dennoch können sich ihre Berechnungen unterscheiden. Ein Grund hierfür kann sowohl eine spezielle Rolle einer Komponente sein, als auch ihre Lage in einem Netzwerk.

Eine Komponente kann eine spezielle Rolle übernehmen. Ein Beispiel hierfür ist die Rolle des Koordinators in einem Netzwerk. Die Rolle des Koordinators könnte jedoch prinzipiell auch von jeder anderen Komponenten des Netzwerks übernommen werden.

Der *verteilte Algorithmus* ist durch das Zusammenspiel aller Teilalgorithmen in einem Netzwerk definiert.

4.2.3 Nicht-Terminierung und Terminierung

Die Terminierung eines sequentiellen Algorithmus ist klassischerweise Teil der Korrektheit des Algorithmus. So definieren Cormen et al. in [Cor+01] – einem Referenzwerk zu sequentiellen Algorithmen – einen Algorithmus als *korrekt*, wenn er für jede (korrekte) Eingabe mit einer korrekten Ausgabe terminiert. Den Referenzwerken verteilter Algorithmen [Lyn96; Ray13; Tel94; Peloo] entnehmen wir hingegen, dass einige verteilte Algorithmen terminieren, während andere fortwährend laufen.

Ein Beispiel für einen *terminierenden* verteilten Algorithmus ist die Wahl einer Komponente als Koordinator im Netzwerk – bekannt als *Leader Election*. Ein Leader-Election-Algorithmus terminiert, wenn eine Komponente des Netzwerks als eindeutiger „Leader“ gewählt ist. Im Allgemeinen ist ein terminierender verteilter Algorithmus

korrekt, wenn er für jede korrekte Eingabe mit der korrekten Ausgabe terminiert.

Ein Beispiel für einen fortlaufenden, also *nicht terminierenden* verteilten Algorithmus, ist der gegenseitige Ausschluss (genannt *Mutex*). Ein Mutex-Algorithmus regelt den gegenseitigen Ausschluss zum Beispiel beim Zugriff auf eine Ressource.

4.3 STAND DER TECHNIK: LAUFZEITVERIFIKATION

Wir gehen von zertifizierenden sequentiellen Algorithmen aus, wie sie in [McC+11] definiert sind. In Abschnitt 4.3.1 geben wir einen allgemeineren Überblick zum Stand der Zertifizierung im Sequentiellen. In Abschnitt 4.3.2 zeigen wir den Stand der Laufzeitverifikation verteilter Systeme auf.

4.3.1 Zertifizierende sequentielle Algorithmen

Aus der Literatur sind über 100 zertifizierende sequentielle Algorithmen bekannt und es gibt eine Vielzahl an Forschungsarbeiten in dem Gebiet [MN98; CW00; NL00; Nam01; Gle03; SYY03; McC04; Wil+04; AR05; NP05; HH05; Cha06; GTW06; BJP06; HK07; BP09; BHS09; KN09; Drä+10; BG11; HC11; McC+11; Alb+12; Sch12; NP12; Cor13; CDH13; NCM14; FMP13; FB14; ZPK14; Hol+16; HNR16; JW17; Jak17]. Es gehören zu jedem Algorithmus auch Datenstrukturen, die er benutzt. Auf dem Gebiet der zertifizierenden Datenstrukturen, die zu zertifizierenden Algorithmen gehören, gibt es weitaus weniger Arbeiten [FM99; GS07; McC+11].

Einige der zitierten Ansätze verfolgen das Konzept des *Proof-carrying-Code*: ein zertifizierender Compiler, der für die Zertifizierung Ergebnisse aus der Typtheorie benutzt [Nec97; NL98]. Auch einige Model-Checker setzen das Konzept der Zertifizierung um [Nam01; SS03; Drä+10; Bey+15; Bey+16; Bey17], ebenso einige Theorembeweiser [NL00; Cor13]. Ein zertifizierender Algortihmus ist außerdem als ein Testorakel nutzbar [Bar+15].

Die *Certification-Trail-Methode* ist verwandt mit dem Konzept eines zertifizierenden sequentiellen Algorithmus [AL94; SM91; BS95; BSM97; SM90; SWM93; SWM95] und entwickelte sich aus der n-Versionen-Programmierung [LH93]. Dabei löst ein Algorithmus ein Problem und generiert zusätzlich einen Zeugen, mit dem ein zweiter, einfacher aufgebauter, Algortihmus die Berechnung gegen prüft. Es gibt jedoch

keine Integration eines Zeugenprädikats und der Zeuge ist zumeist eine Sequenz mit Informationen zur Berechnung.

Auch die Methode des *Program Checking*, für die Blum einen Turing-Award erhielt, entwickelte sich aus der n-Versionen-Programmierung.[1] Beim Program Checking interagiert ein Checker-Programm mit einem Programm, um Informationen über die Korrektheit des Programms zu erhalten [Rub90; Blu93; BK95; FGY96; BW97; Erg+98]. Es gibt einige Gemeinsamkeiten mit der Zertifizierung, doch auch Unterschiede. Der Checker arbeitet probabilistisch und verifiziert eine Wahrscheinlichkeit der Korrektheit des Programms und nicht einer Instanz. Blum forschte auch zu einer Erweiterung für Datenstrukturen [Blu+94]. Darüber hinaus inspirierte Program Checking die Entwicklung probabilistisch prüfbarer Beweise [AS98].

4.3.2 Laufzeitverifikation verteilter Systeme

Ideen der n-Versionen-Programmierung und darüber hinaus finden wir auch bei Methoden zur verteilten Verarbeitung großer Datenmengen (MapReduce) wieder [DSMS07; MSF11; HZW12]. So gibt es zum Beispiel eine Abstimmung aller Komponenten zur Korrektheit oder aber es werden Zwischenergebnisse strukturiert in einem Baum (Merkle-Baum) gesammelt, der dann als Beweis für eine korrekte Berechnung dient.

Wir haben in Kapitel 1 bereits darauf hingewiesen, dass es zwar viele Ansätze zur Laufzeitverifikation verteilter Systeme gibt, dabei jedoch wenige Ansätze zur verteilten Laufzeitverifikation. In [FPS18] wird ein Überblick über Ansätze zur *verteilten* Laufzeitverifikation verteilter Systeme geboten. Üblicherweise wird bei diesen Ansätzen ein verteiltes System so instrumentalisiert, dass ein sequentieller Monitor während der Berechnung eine Sicherheitseigenschaft (ausgedrückt als temporal logische Formel) zentral für das System prüfen kann. Es gibt dabei kein Zeugenprädikat und es geht dabei auch nicht um die Korrektheit eines Eingabe-Ausgabe-Paars.

Eine andere interessante Richtung ist aus Sicht der Zertifizierung das Konzept der Selbst-Stabilisierung in verteilten Systemen [Dolo0]. Ein selbst-stabilisierender Algorithmus gelangt nach endlicher Zeit in einen korrekten Zustand unabhängig davon aus welchem Zustand heraus er startet. Das Fehlermodell gegen das die Selbst-Stabilisierung tolerant ist, ist ein verteiltes System in dem anfänglich Fehler „diverser Art“ auftreten können, das jedoch nach endlicher Zeit fehlerfrei ist. Es handelt sich somit nicht um Zertifizierung. Das erkennen wir leicht daran, dass eine simple Variante einen terminierenden verteil-

1 Webseite mit Informationen zum Turing-Award für Manuel Blum: https://amturing.acm.org/award_winners/blum_4659082.cfm

ten Algorithmus selbst-stabilisierend zu gestalten, dessen mehrfache Ausführung ist. Dennoch gibt es auch Gemeinsamkeiten und das liegt daran, dass für die Selbst-Stabilisierung häufig ein Mechanismus genutzt wird, der bezeugt, dass der Zustand noch nicht oder aber bereits korrekt ist.

Eine weitere interessante Richtung ist die Komplexitätstheorie verteilter Algorithmen. Dafür ist zunächst der folgende Zusammenhang zwischen Zertifizierung und der Komplexität sequentieller Algorithmen interessant. Der Begriff des Zeugen ist aus der Komplexitätstheorie sequentieller Algorithmen bekannt: Jedes in nicht-deterministischer polynomieller Zeit (NP) gelöste Problem ist in deterministischer polynomieller Zeit (P) prüfbar mit einem Zeugen, der die Berechnung selbst ist – ein Pfad in dem nicht-deterministischen Berechnungsbaum mit polynomieller Tiefe [Gol10]. Tarjan formulierte deswegen auch das Verifikationsproblem. Dabei ist das Ziel, durch die Betrachtung der Beziehung zwischen Berechnung und Verifikation, auch Beziehungen zwischen Komplexitätsklassen herzustellen [Tar79].

Inspiriert durch die sequentielle Komplexitätstheorie wird das Problem der verteilten Verifikation betrachtet, um Beziehungen zwischen verteilten Komplexitätsklassen herzustellen, die der zwischen NP und P ähneln [KK06; KKP10; DS+11; FF16; BF17; OPR17]. Dabei wird auch untersucht, wie sich Informationen zur Berechnung (Zeugen) auf die Beziehung der Komplexitätsklassen auswirken, ganz analog zu den Betrachtungen zu P und NP. Zum Beispiel ermöglichen bestimmte Informationen der Berechnung (locally checkable proof) eine Verifikation in konstanter Kommunikationszeit (lokaler verteilter Algorithmus) [GS11; Suo13; AFP13; FKP13].

Wir können bei den Betrachtungen zur Komplexität also sowohl bei verteilten als auch sequentiellen Algorithmen somit Gemeinsamkeiten zur Zertifizierung finden. Das Ziel ist jedoch ein anderes und so gibt es auch keine Zeugenprädikate, keine Integration eines Checkers, kein passendes Fehlermodell und eben keine softwaretechnische Perspektive. Ein Beispiel illustriert den Unterschied: Eine 2-Färbung eines Netzwerks (die keine Bipartition ist) dient bei der verteilten Verifikation dazu, zu verifizieren, dass das Netzwerk nicht bipartit ist. Das ist jedoch kein Zeuge im Sinne einer Zertifizierung, da wir dem verteilten Algorithmus nicht vertrauen und der berechnete Zeuge falsch sein kann. Bei uns wäre ein ungerader Kreis (wie im Sequentiellen) ein Zeuge; dieser ist wiederum mit den Prädikaten der verteilten Verifikation nicht beschreibbar.

Ein weiterer interessanter Ansatz wird in [BS01] beschrieben – eine ausführbare Spezifikationssprache für Komponenten im Sinne der Abstract State Machines [Gur95]. Die Spezifikation läuft nebenläufig zur eigentlichen Implementierung und ein Abweichen im Verhalten wird erkannt. Die Verifikation zur Laufzeit sagt dabei nichts über

die allgemeine Korrektheit einer Komponente aus, aber sie belegt die Korrektheit des aktuellen Ablaufs. Es handelt sich also um eine Instanzverifikation, jedoch wird dabei nicht das gesamte Verhalten betrachtet, sondern jede Komponente für sich.

Teil III

ZERTIFIZIERENDE TERMINIERENDE VERTEILTE ALGORITHMEN

Wir stellen in diesem Teil eine Methode vor, das Konzept *zertifizierender sequentieller Algorithmen* auf *verteilte Algorithmen* zu übertragen. Bei der Übertragung berücksichtigen wir die beiden motivierten Ziele: zum einen nah am ursprünglichen Konzept zu bleiben und zum anderen die Gegebenheiten verteilter Systeme zu berücksichtigen. Als Resultat einer Abwägung der beiden, teilweise gegensätzlichen, Ziele erhalten wir eine Klasse zertifizierender verteilter Algorithmen, die terminieren, ihr *verteiltes* Eingabe-Ausgabe-Paar verifizieren, ein *verteilbares* Zeugenprädikat besitzen und einen *verteilten* Zeugen je Eingabe-Ausgabe-Paar berechnen.

Zunächst beschäftigen wir uns in Kapitel 5 damit, wie wir ein Problem spezifizieren, das ein verteilter Algorithmus in einem Netzwerk lösen soll. In Kapitel 6 geben wir zur Intuition ein erstes Beispiel eines zertifizierenden verteilten Algorithmus, bevor wir nach und nach die entsprechenden Konzepte einführen. Im selbigen Kapitel führen wir bereits verteilbare Zeugenprädikate ein. In Kapitel 7 beschäftigen wir uns eingehend mit verteilten Zeugen. Schließlich definieren wir in Kapitel 8 die resultierende Klasse zertifizierender verteilter Algorithmen. In Kapitel 9 stellen wir sieben Fallstudien zertifizierender verteilter Algorithmen zur Illustration vor.

5 SPEZIFIKATION DES EINGABE-AUSGABE-VERHALTENS

Ein verteilter Algorithmus wird entworfen, um ein bestimmtes Problem in einem verteilten System zu lösen. Wir beschäftigen uns in diesem Kapitel damit, wie wir ein solches Problem spezifizieren.

In Abschnitt 5.1 erläutern wir zunächst einige Kerngedanken verteilter Algorithmen. Diese Kerngedanken liefern uns einen Anhaltspunkt dafür, wann eine Zertifizierung zu den Gegebenheiten eines verteilten Systems passt. Im Anschluss führen wir in Abschnitt 5.2 unser Netzwerkmodell ein.

In Abschnitt 5.3 diskutieren wir den Einfluss der Terminierung auf die Zertifizierung eines verteilten Algorithmus. Da wir uns auf terminierende verteilte Algorithmen beschränken, stellen wir in Abschnitt 5.4 unsere Modellierung verteilter Eingabe-Ausgabe-Paare vor.

In Abschnitt 5.5 zeigen wir auf, wie wir ein Problem über Eingabe-Ausgabe-Paare in einem Netzwerk spezifizieren und legen somit die Grundlage für eine Laufzeitverifikation.

Wir erläutern durchgehend über die verschiedenen Themen hinweg unsere Abwägung für die beiden Ziele bei einer Übertragung des Konzepts zertifizierender sequentieller Algorithmen auf verteilte Algorithmen.

5.1 KERNGEDANKEN VERTEILTER ALGORITHMEN

Wir stellen in diesem Abschnitt drei Kerngedanken verteilter Algorithmen vor, die wir aus Referenzwerken verteilter Algorithmen abgeleitet haben [Tel94; Lyn96; Pel00; AW04; Rei10; Erc13; Ray13; Gho14]. Die Kerngedanken beschreiben dabei erstrebenswerte Eigenschaften für verteilte Algorithmen, sind jedoch keine harte Richtlinie dafür, was ein verteilter Algorithmus ist. Wir stellen die Kerngedanken der Verteiltheit (Abschnitt 5.1.1), Gleichheit (Abschnitt 5.1.2) und Lokalität (Abschnitt 5.1.3) im Folgenden vor.

5.1.1 Verteiltheit

Ein verteilter Algorithmus soll *verteilt* auf allen Komponenten eines Netzwerks ausgeführt werden; das heißt, dass alle Komponenten tatsächlich an der Berechnung beteiligt sind und einen Beitrag leisten. Im Umkehrschluss sollen also nicht ein paar Komponenten eines Netzwerks ein Problem im Grunde sequentiell lösen. In diesem Fall sprechen wir von einer Zentralisierung, die vermieden werden soll.

Allerdings gibt es bei vielen verteilten Algorithmen eine teilweise Zentralisierung dadurch, dass eine Komponente für einen verteilten Algorithmus eine Sonderrolle zur Koordination einnimmt. Zum Beispiel wenn eine Komponente die Wurzel eines Spannbaums im Netzwerk ist, über die Berechnungen aller Komponenten zusammengeführt werden. Die Wurzel hat dann eine zentralisierende Rolle. Ohne ein gewisses Maß an Zentralisierung könnten jedoch viele Probleme nicht durch einen verteilten Algorithmus gelöst werden.

5.1.2 Gleichheit

Für einen verteilten Algortihmus sollen alle Komponenten eines Netzwerks *gleich* sein; das heißt, dass die Teilalgorithmen so aufgebaut sind, dass die Berechnungslast gleichermaßen auf die Komponenten verteilt ist. Dabei sind auch die Teileingabe und Teilausgabe einer jeden Komponente gleich aufgebaut, auch wenn sie sich gegebenenfalls in ihren Werten unterscheiden.

Allerdings sind Komponenten in den meisten Modellen und für die meisten verteilten Algorithmen nicht vollkommen gleich. So unterscheiden sich Komponenten zum Beispiel durch ihre Position im Netzwerk. Auch eine Sonderrolle führt zu einer Unterscheidung und diese soll deswegen auch immer möglichst klein gehalten werden. In unserem Netzwerkmodell unterscheiden sich Komponenten darüber hinaus auch durch ihre IDs.

5.1.3 Lokalität

Während eines verteilten Algorithmus sollen die Komponenten *lokal* kommunizieren. Ein hartes Kriterium hierfür ist, dass der Kommunikationsradius einer Komponente durch eine Konstante beschränkt ist, die von dem zu lösenden Problem, nicht aber dem Netzwerk abhängt. In der Literatur werden solche Algorithmen auch als *lokale Algorithmen* bezeichnet. Es wurde sich eingehend damit beschäftigt, was mit lokalen Algorithmen berechnet werden kann [NS95; Fra+13; Suo13], und was nicht [KMW04], sowie mit Betrachtungen zu deren Komplexität [FKP13].

Neben diesem harten Kriterium zur Lokalität, finden wir es aber auch als weiches Ziel in der Literatur wieder. Obwohl sich viele Probleme nicht mit einem lokalen Algorithmus lösen lassen, wird auch für verteilte Algorithmen, die solche Probleme lösen, ein möglichst hoher Grad an Lokalität angestrebt. So beschreibt Peleg einen *globalen* Algorithmus nicht als Gegenteil eines lokalen Algorithmus, sondern als einen bei dem die vorhandene Lokalität einer Berechnung nicht ausgenutzt wurde [Peloo]:

> Informally speaking, a "global" algorithm is one that makes no attempt to restrict its activities to the specific regions of the network directly related to its task.

5.1.4 Einordnung der Kerngedanken

Die verteilten Algorithmen der Referenzwerke [Tel94; Lyn96; Peloo; AW04; Rei10; Erc13; Ray13; Gho14] richten sich nach den Kerngedanken, erfüllen sie aber selten in Gänze. Das liegt daran, dass die Kerngedanken teilweise auch gegeneinander abgewägt werden müssen.

Zusätzlich liegt es daran, dass sie teilweise auch mit anderen wünschenswerten Eigenschaften abgewägt werden müssen, die sich zum Beispiel durch ein spezielles Einsatzgebiet ergeben. So eine Eigenschaft könnte eine bestimmte Laufzeit oder Speicherkomplexität sein.

Darüber hinaus ist ein verteilter Algorithmus, der alle Kerngedanken in Gänze erfüllt, häufig gar nicht möglich. Zum Beispiel kann keine Symmetrie gebrochen werden, wenn alle Komponenten komplett gleich und damit nicht unterscheidbar sind. Diese Beobachtung ist zum Beispiel interessant für den Kerngedanken der Gleichheit und unsere Annahme von IDs für die Komponenten.

5.2 NETZWERKMODELL

Ein verteilter Algorithmus ist dafür entworfen, auf einem verteilten System zu laufen. Wir betrachten in dieser Arbeit Netzwerke und begründen unsere Wahl in Abschnitt 5.2.1.

In Abschnitt 5.2.2 führen wir ein graphenbasiertes Modell für Netzwerke ein. Dabei betrachten wir ID-basierte Netzwerke, die wir in Abschnitt 5.2.3 erläutern. Da wir außerdem asynchrone Netzwerke untersuchen, erläutern wir die Arbeitsweise der Komponenten in einem asynchronen System in Abschnitt 5.2.4.

In Abschnitt 5.2.5 diskutieren wir unsere Darstellung verteilter Algorithmen und in Abschnitt 5.2.6 unser Fehlermodell.

5.2.1 Netzwerke für die Betrachtung verteilter Systeme

Wir entscheiden uns als Klasse verteilter Systeme Netzwerke und nicht Systeme mit geteiltem Speicher zu untersuchen. Die Komponenten eines Netzwerks weisen einen größeren Grad an Unabhängigkeit auf im Vergleich zu den Komponenten eines Systems mit geteiltem Speicher.

Peleg beschreibt den Unterschied eines verteilten Systems mit geteiltem Speicher im Vergleich zu einem Netzwerk in [Peloo] wie folgt:

> [...] the shared memory model is a commonly studied one, it is more suitable for discussing very tightly coupled concurrent or parallel systems [...]

Wir erachten ein Netzwerk als einen besseren Repräsentanten für die speziellen Herausforderungen, die ein verteiltes System an die Laufzeitverifikation mit einem zertifizierenden verteilten Algorithmus darstellt. Wir erwarten, dass wir mit unseren Betrachtungen auf Basis von Netzwerken allen Besonderheiten begegnen, denen wir auch bei einer Betrachtung auf Basis von verteilten Systemen mit geteiltem Speicher begegnen würden.

5.2.2 Modellierung des Netzwerks

Die *Topologie* eines Netzwerks legt die möglichen Kommunikationswege fest, indem sie beschreibt, welche Komponenten durch Kanäle benachbart sind. Wie üblich für Netzwerke, modellieren wir die Topologie eines Netzwerk als einen zusammenhängenden ungerichteten Graphen [Ray13]. Ein Knoten des Graphen repräsentiert eine Komponente des Netzwerks und eine Kante einen bidirektionalen Nachrichtenkanal.

Wir sprechen oft auch verkürzt von einem Netzwerk anstatt von dessen Topologie, wenn der Kontext eindeutig ist. Wir werden meist von Komponenten und Kanälen und nicht von Knoten und Kanten sprechen, denn eine Komponente kann rechnen und ein Knoten nicht. Das ist auch deswegen wichtig, weil wir uns auch auf zertifizierende sequentielle Graphalgorithmen beziehen, wo mit einem Knoten eben keine Komponente gemeint ist. An manchen Stellen weichen wir von dieser Sprechweise jedoch ab, weil wir über etablierte graphentheoretische Konzepte im Kontext von Netzwerken sprechen und die Benennung sonst verwirrend sein könnte. Zum Beispiel sagen wir Elternknoten und nicht Elternkomponente.

5.2.2.1 *Benötigte Konzepte aus der Graphentheorie*

Wir definieren in Kürze die benötigten Konzepte aus der Graphentheorie, wie wir sie für diese Arbeit verwenden. Informierte Leser:innen können diesen kurzen Abschnitt überspringen.

Definition 2 (Ungerichteter Graph)**.** *Sei* V *eine nicht-leere, endliche Menge. Sei* $E \subseteq V \times V$ *eine Menge.*

(i) Das Tupel $G = (V, E)$ *ist ein* Graph *mit den Knoten* V *und Kanten* E*.*

(ii) G *ist* ungerichtet*, falls für jede Kante* $(v_1, v_2) \in E$ *gilt* $(v_2, v_1) \in E$*.*

Definition 3 (Teilgraph)**.** *Sei* $G = (V, E)$ *ein ungerichteter Graph. Der Graph* $T = (V_1, E_1)$ *ist Teilgraph von* G*, falls* $V_1 \subseteq V$ *und* $E_1 \subseteq E$*.*

Wir sagen außerdem, dass ein Teilgraph $T = (V_1, E_1)$ von $G = (V, E)$ durch seine Knotenmenge V_1 *induziert* ist, falls für alle $u, v \in V_1$ gilt: wenn $(u, v) \in E$, dann $(u, v) \in E_1$.

Definition 4 (Pfad)**.** *Sei* $G = (V, E)$ *ein Graph. Ein* Pfad *ist eine Knotenfolge* $v_0, v_1, ..., v_l$ *von verschiedenen Knoten* $v_i \in V$ *mit* $0 < i \leqslant l$*, sodass gilt* $(v_{i-1}, v_i) \in E$*.*

Die *Länge* eines Pfads $v_0, v_1, ..., v_l$ ist die Anzahl l seiner Kanten. Wir nennen den Pfad auch kurz v_0-v_l-Pfad. In einem ungerichteten Graphen gibt es zu einem v_0-v_l-Pfad auch immer einen v_l-v_0-Pfad mit umgedrehter Knotenfolge. Wir betrachten beide Knotenfolgen als den gleichen Pfad.

Definition 5 (Zusammenhang)**.** *Sei* $G = (V, E)$ *ein ungerichteter Graph.* G *ist* zusammenhängend*, falls es für jedes Knotenpaar* $(v_1, v_2) \in V$ *einen* v_1-v_2*-Pfad gibt.*

5.2.2.2 *Definition eines Netzwerks*

Wir gehen von einem *statischen* Netzwerk aus; das heißt, dass sowohl die Anzahl der Komponenten als auch die Topologie des Netzwerks unverändert bleiben. Wir definieren ein Netzwerk deswegen wie folgt.

Definition 6 (Netzwerk)**.** *Sei* $N = (V, E)$ *ein zusammenhängender, ungerichteter Graph.* N *ist ein* Netzwerk *mit* Komponenten V *und* Kanälen E*.*

Jede Komponente hat eine andere Position im Netzwerk und somit zum Beispiel eine unterschiedliche Anzahl an Nachbarn oder auch eine unterschiedliche Entfernung zu einer anderen Komponente. Die Position einer Komponente kann somit Einfluss auf die Kommunikation und die lokale Berechnung einer Komponente haben.

Genauso unterscheiden sich die Nachrichtenkanäle nicht von einander. In einigen Fällen sind Kanäle jedoch mit unterschiedlichen Kosten

assoziiert. In diesem Fall modellieren wir ein Netzwerk als einen zusammenhängenden, ungerichteten Graphen mit *gewichteten* Kanten. Das Gewicht einer Kante repräsentiert dann die Kosten eines Kanals. Für ein Netzwerk $N = (V, E)$ beschreibt die Funktion $\mathrm{cost} : E \to \mathbb{R}_+$ die Kosten seiner Kanäle. (Hierbei ist $\mathbb{R}_+$ die Menge der positiven reellen Zahlen ohne die Null.)

5.2.2.3 k-*Nachbarschaft und Lokalität*

Zwei Komponenten sind k-*Nachbarn*, wenn die Länge eines kürzesten Pfades zwischen den Komponenten die Länge k hat. Die k-*Nachbarschaft* einer Komponente sind alle l-Nachbarn mit $l \leqslant k$; dabei sind die 1-Nachbarn die Nachbarn der Komponente, wie wir sie bisher definiert haben.

Wir benutzen des Öfteren die Begriffe lokal und global in einem Netzwerk. Während sich *lokal* auf eine Komponente oder eine k-Nachbarschaft in einem Netzwerk bezieht, bezieht sich global auf das gesamte Netzwerk.

5.2.3 ID-basiertes Netzwerk

Netzwerke werden danach unterschieden, ob die Komponenten Identifikationsnummern (IDs) haben. Es gibt zahlreiche Modelle, die Folgenden werden am häufigsten genutzt:

- Netzwerke, in denen es keine IDs gibt [AR09a]. Als Konsequenz kann eine Komponente ihre Nachbarn nicht unterscheiden. Nur wenige gängige verteilte Algorithmen sind für solche Netzwerke ausgelegt.
- Netzwerke, in denen die Nachbarn einer jeden Komponente eindeutige IDs haben. Eine Komponente kann also ihre Nachbarn von einander unterscheiden [Hel+12].
- ID-basierte Netzwerke, in denen jede Komponente eine eindeutige ID besitzt [Ray13; Peloo; Gho14].

Die meisten Referenzwerke zu verteilten Algorithmen gehen von ID-basierten Netzwerken aus und entsprechend sind die meisten verteilten Algorithmen für ID-basierte Netzwerke ausgelegt [Ray13; Peloo; Gho14]. Wir nehmen deswegen ID-basierte Netzwerke an.

Weiterhin sei über den IDs eine Ordnung definiert. Wir nehmen für ein Netzwerk $N = (V, E)$ an, dass die ID einer jeden Komponente ihrer Benennung entspricht; das heißt, für alle $v \in V$ gilt $\mathrm{id}_v = v$. Wir bezeichnen eine Komponente v deswegen gleichwertig mit v als auch mit ihrer ID id_v. Die ID einer Komponente ist fix.

Jede Komponente kennt ihre Nachbarschaft. Wir nehmen an, dass jede Komponente neben ihrer eigenen ID, die IDs ihrer Nachbarn kennt und, sofern vorhanden, die Kosten ihrer Kanäle. Auch hierbei handelt es sich um eine übliche Annahme [Gho14; Ray13; Lyn96; Peloo; AW04].

5.2.4 Asynchrone Netzwerke: Annahmen

5.2.4.1 *Kanäle*

Wir nehmen an, dass die Zeit, die eine Nachricht braucht, um über einen Kanal gesendet zu werden, immer endlich, aber unbestimmt lang ist. Diese Annahme ist in der Literatur üblich [Gho14; Ray13; Peloo]. Wir nehmen außerdem an, dass ein Kanal eine unbeschränkte Kapazität hat; das heißt, er kann beliebig viele Nachrichten und beliebig große Nachrichten enthalten. Diese Annahme macht beispielsweise auch Raynal in seinem Referenzwerk [Ray13].

In der Literatur wird manchmal auch eine maximal mögliche Größe einer Nachricht angenommen. Das ist vor allem dann der Fall, wenn die Anzahl der Nachrichten für eine Komplexitätsanalyse betrachtet wird [Peloo].

Wir verzichten an dieser Stelle auf eine Größenbeschränkung. Allerdings weisen wir daraufhin, dass wir bei der Gestaltung einer zertifizierenden Variante eines verteilten Algorithmus stets von einem bereits bekannten verteilten Algorithmus ausgehen. Die Nachrichten, die zusätzlich für die zertifizierende Variante nötig sind, sind dann verhältnismäßig zu der Größe der Nachrichten des ursprünglichen Algorithmus.

5.2.4.2 *Komponenten*

Die Arbeitsweise einer Komponente sieht wie folgt aus. Wenn sie eine Nachricht erhält, dann verarbeitet sie die Nachricht, indem sie Berechnungen lokal ausführt. Je nach Ergebnis ihrer Berechnung sendet die Komponente eigene Nachrichten an ihre Nachbarn. Eine Komponente verarbeitet eine Nachricht nach der anderen. Die Verarbeitung einer Nachricht wird also nicht unterbrochen, wenn die Komponente eine neue Nachricht empfängt. Die neu empfangene Nachricht wird in einem Nachrichtenpuffer der Komponente zwischengelagert. Die Nachricht wird abgearbeitet sobald alle vorigen Nachrichten im Puffer abgearbeitet sind. Der Puffer arbeitet also nach dem First-In-First-Out-Prinzip.

Eine Komponente ist während der Abarbeitung einer Nachricht in einem *aktiven* Zustand. Zur Abarbeitung gehören die lokalen Berech-

nungen der Komponente, sowie gegebenenfalls das Senden von Nachrichten an ihre Nachbarn.

Nach der Abarbeitung einer Nachricht ist die Komponente in einem *passiven* Zustand. In diesem Zustand führt eine Komponente keine lokalen Berechnung aus und verschickt auch keine Nachrichten. Eine Komponente wird erst wieder aktiv, wenn sie eine Nachricht erhält. Die Unterscheidung zwischen den Zuständen aktiv und passiv ist interessant für die Betrachtungen zur Terminierung in Abschnitt 5.3 (vgl. mit [Ray13, S. 368]).

Die Eingabe eines verteilten Algorithmus besteht aus den Eingabewerten für das zu lösende Problem, sowie dem initialen Wissen der Komponenten zum Netzwerk. Mindestens eine Komponente muss anfangs so initialisiert sein, dass sie aktiv ist, damit sie den verteilten Algorithmus in Gang setzen kann.

5.2.5 Darstellung eines verteilten Algorithmus

Als Darstellungsform eines verteilten Algorithmus in dieser Arbeit wählen wir eine textuelle Beschreibung. Unsere Wahl hat die folgenden Gründe. Wir betrachten etablierte, bekannte verteilte Algorithmen, deren Eigenschaften in der Literatur bereits gut untersucht sind. Wir untersuchen diese Algorithmen auf keine Eigenschaften, die ein spezielles Modell für die Analyse erfordern würden.

Wir ändern die bereits bekannten verteilten Algorithmen ab, um sie zertifizierend zu gestalten. Die zertifizierende Variante berechnet dann zusätzlich einen Zeugen. Die Berechnung des Zeugen soll nach Möglichkeit so in die Berechnung des ursprünglichen verteilten Algorithmus integriert sein, dass von den Informationen, die während der Berechnung ohnehin abfallen, profitiert wird. Das heißt im Idealfall werden sogar lediglich Informationen während der Berechnung für den Zeugen gesammelt und nicht einmal zusätzlich berechnet. Manchmal ist allerdings auch etwas mehr zu tun, um den verteilten Algorithmus zertifizierend zu gestalten.

Die relevante Eigenschaft eines zertifizierenden Algorithmus ist, dass er sein Eingabe-Ausgabe-Paar zur Laufzeit verifiziert. Uns ist kein Modell für verteilte Algorithmen bekannt, dass speziell die Verifikation dieser Eigenschaft erleichtern würde. In Teil v stellen wir außerdem ein Framework für den Beweisassistenten `COQ` für die Verifikation genau dieser Eigenschaft vor, sodass wir nicht auf Beweise auf Papier vertrauen müssen.

Ein weiterer Grund ist, dass wir damit die gleiche Darstellungsform wählen, wie sie für zertifizierende sequentielle Algorithmen üblich ist [McC+11]. Darüber hinaus ist es auch eine der gängigen Darstellungsformen verteilter Algorithmen [Ray13; AW04; Pel00; Gho14].

5.2.6 Fehlermodell

Es gibt zahlreiche Fehlermodelle für verteilte Systeme [Lyn96; AW04; Pelo0; Gho14] und die Entwicklung fehlertoleranter verteilter Algorithmen ist ein eigenes großes Forschungsgebiet [AW04; Gho14]. Wir möchten das Fehlermodell, das den Betrachtungen zur Zertifizierung sequentieller Algorithmen zugrunde liegt übernehmen.

Ein zertifizierender sequentieller Algorithmus ist fehlertolerant für die folgenden Fehler [McC+11]:

- fehlerhafter Algorithmus,
- fehlerhafte Implementierung des Algorithmus und
- fehlerhafte Ausführung des Algorithmus.

5.2.6.1 *Verteilter Algorithmus nicht vertrauenswürdig*

Wir interessieren uns für ein Fehlermodell, dass diese Fehler in einem Netzwerk widerspiegelt, sodass ein zertifizierender *verteilter* Algorithmus in derselben Weise fehlertolerant ist, wie ein zertifizierender *sequentieller* Algorithmus.

Wir nehmen dafür an, dass ein verteilter Algorithmus nicht vertrauenswürdig ist. In unserem Fehlermodell können deswegen die folgenden Fehler auftreten:

- fehlerhafter Teilalgorithmus einer Komponente,
- fehlerhafte Implementierung eines Teilalgorithmus,
- fehlerhafte Ausführung eines Teilalgorithmus.

Wir wählen dieses Fehlermodell, da es die Fehler, gegen die ein zertifizierender sequentieller Algorithmus tolerant ist, verhältnismäßig direkt in ein Netzwerk übersetzt.

Dabei ist zu beachten, dass ein Fehler häufig auch andere Fehler simuliert. So nehmen wir zwar fehlerfreie Nachrichtenkanäle an, aber ein fehlerhafter Teilalgorithmus einer Komponente simuliert auch einige Fehler für Nachrichtenkanäle.

Nehmen wir beispielsweise an, eine Komponente erhält eine fehlerhafte Nachricht. Diese Nachricht wurde gegebenenfalls über eine Verbindung über einige Komponenten und Kanäle im Netzwerk vom Sender zum Empfänger weitergeleitet. Für die Empfänger-Komponente ist nicht unterscheidbar, ob die fehlerhafte Nachricht durch eine Korruption eines Nachrichtenkanal oder einen fehlerhaften Teilalgorithmus einer Komponente zu Stande kam.

5.2.6.2 *Abgrenzung zum byzantinischen Fehlermodell*

Ein Fehlermodell, in dem die Komponenten nicht vertrauenswürdig sind, wird häufig als *byzantinisch* bezeichnet [LM07]. In der Literatur wird oft zusätzlich noch angenommen, dass Komponenten das System mit oder ohne Benachrichtigung verlassen können. Byzantinische Fehler sind schwierig zu beherrschen. Es gibt diverse Unmöglichkeitsresultate für ein byzantinisches Fehlermodell [LM07].

Obwohl unser Fehlermodell auf den ersten Blick nach einem byzantinischen Fehlermodell aussieht, unterscheidet es sich. Bei einem zertifizierenden sequentiellen Algorithmus gehen wir zwar davon aus, dass der Algorithmus fehlerhaft sei kann, der Checker jedoch korrekt ist. Das heißt der Algorithmus des Checkers, dessen Implementierung und dessen Ausführung sind fehlerfrei.

Analog zum Fehlermodell für zertifizierende sequentielle Algorithmen werden auch wir korrekte Checker annehmen und entsprechend diskutieren, wie für diese Korrektheit garantiert werden kann. An dieser Stelle ist jedoch erst einmal wichtig festzuhalten, dass wir für unsere Betrachtungen zur Fehlertoleranz damit eine weitere Annahme haben. Somit unterscheidet sich unser Fehlermodell grundlegend von einem „klassischen“ byzantinischen Fehlermodell.

5.3 TERMINIERUNG UND ZERTIFIZIERUNG

Für unser Teilziel, nah am Konzept eines zertifizierenden sequentiellen Algorithmus zu bleiben, beschränken wir uns an dieser Stelle auf terminierende verteilte Algorithmen.

Wir diskutieren deshalb was Terminierung in einem Netzwerk bedeutet (Abschnitt 5.3.1) und welchen Einfluss sie auf die Zertifizierung verteilter Algorithmen hat (Abschnitt 5.3.3).

5.3.1 Terminierung in einem Netzwerk

Bei einem *sequentiellen* Algorithmus gibt es nur *eine* rechnende Einheit, die den Algorithmus sequentiell ausführt. Wenn der Algorithmus terminiert, weiß die ausführende Einheit das. Im Gegensatz dazu müssen wir bei der Ausführung eines verteilten Algorithmus in einem Netzwerk alle rechnenden Einheiten, also die Komponenten des Netzwerks, betrachten.

Ein verteilter Algorithmus ist terminiert, wenn das Netzwerk in einem Zustand ist, in dem alle Komponenten passiv sind und keine Nachrichten mehr unterwegs sind, siehe [Ray13, Kapitel 14]. Wenn die

Terminierung einmal eintritt, dann bleibt sie erhalten. Das liegt daran, dass wenn alle Komponenten passiv sind, keine weiteren Nachrichten mehr verschickt werden und dass wenn alle Kanäle leer sind, auch keine Komponente mehr aktiv wird. Die Terminierung kann sukzessive erreicht werden, indem die Komponenten nach und nach passiv und die Kanäle nach und nach leer werden.

5.3.2 Festgestellte Terminierung

Wir unterscheiden, wie in der Literatur üblich, zwischen einer Terminierung und einer festgestellten Terminierung des verteilten Algorithmus. Bei letzterem weiß entsprechend jede Komponente, dass die Terminierung des verteilten Algorithmus eingetreten ist. Das erste Mal wurde das Problem der verteilten Feststellung der Terminierung von Francez [Fra80] und unabhängig davon auch von Dijkstra und Scholten [DS80] betrachtet. Seitdem wurden zahlreiche verteilte Algorithmen zur Feststellung der Terminierung entwickelt. In [Ray13, Kapitel 14] werden einige vorgestellt.

Viele verteilte Algorithmen zur Feststellung der Terminierung sind jeweils auf spezielle Netzwerkmodelle zugeschnitten, um besonders effizient zu sein. Gemeinsam haben diese Algorithmen, dass die Terminierung an sich nicht durch die Feststellung der Terminierung unendlich hinausgezögert werden kann.

Die gängigen verteilten Algorithmen zur Feststellung der Terminierung arbeiten zweigliedrig [Ray13, Kapitel 14]. Zum einen führt jede Komponente über ihren lokalen Zustand buch, indem sie beispielsweise die Anzahl ihrer gesendeten und empfangenen Nachrichten zählt. Zum anderen kommunizieren die Komponenten, um sich gemeinsam ein Bild über den globalen Zustand der Ausführung des verteilten Algorithmus zu machen. Letzteres ist vor allem bei einem asynchronen Netzwerk herausfordernd. In [Ray13, Kapitel 14] stellt Raynal einen generischen Algorithmus zur Feststellung der Terminierung vor. Generisch ist dieser insofern, dass er „lediglich“ einen Koordinator unter den Komponenten voraussetzt.

Wir nehmen für terminierende verteilte Algorithmen an, dass die Komponenten die Terminierung feststellen, dafür kann jeder Algorithmus, der zu unserem Netzwerkmodell passt und als zusätzliche Voraussetzung höchstens einen Koordinator fordert, benutzt werden.

5.3.3 Einfluss der Terminierung auf die Zertifizierung

Ein zertifizierender sequentieller Algortihmus verifiziert seine Eingabe-Ausgabe-Paare zur Laufzeit. Für terminierende verteilte Algo-

rithmen gibt es ein *verteiltes* Eingabe-Ausgabe-Paar. Terminierende verteilte Algorithmen bieten also eine verhältnismäßig direkte Möglichkeit das Konzept eines zertifizierenden sequentiellen Algorithmus zu übertragen. Da terminierende verteilte Algorithmen eine große Klasse verteilter Algorithmen darstellen, passt eine Betrachtung der Zertifizierung für terminierende verteilte Algorithmen auch zu den Gegebenheiten verteilter Systeme [Peloo].

Wir beschränken uns deshalb für diesen Teil der Arbeit auf terminierende verteilte Algorithmen. In Teil vi diskutieren wir jedoch auch die Zertifizierung verteilter Algorithmen, die nicht terminieren.

5.3.3.1 *Fehlerhafte Feststellung der Terminierung*

Nach unserem Fehlermodell kann sowohl ein verteilter Algorithmus nicht terminieren, obwohl er sollte, als auch die Feststellung der Terminierung des verteilten Algorithmus fehlerhaft sein. Wenn der verteilte Algorithmus nicht terminiert, dann kommt es nicht zur Laufzeitverifikation.

Wird eine Terminierung fälschlicherweise festgestellt, dann ist die verteilte Ausgabe, die zu dem Zeitpunkt vorliegt entweder trotzdem korrekt für die Eingabe oder inkorrekt. Für die Zertifizierung benutzen wir die verteilte Ausgabe, die zur festgestellten Terminierung vorliegt.

5.4 MODELLIERUNG: EINGABE-AUSGABE-PAAR

Ein terminierender verteilter Algorithmus startet seine Berechnung ausgehend von einer Eingabe und terminiert mit einer Ausgabe. Im Gegensatz zum sequentiellen Algorithmus liegt bei einem verteilten Algorithmus sowohl die Eingabe als auch die Ausgabe im Netzwerk *verteilt* vor. Wir stellen in diesem Abschnitt eine Modellierung des verteilten Eingabe-Ausgabe-Paars eines terminierenden verteilten Algorithmus vor.

In Abschnitt 5.4.1 besprechen wir das Eingabe-Ausgabe-Verhalten einer Komponente für terminierende verteilte Algorithmen. In Abschnitt 5.4.2 erläutern wir, warum wir ein Verteilen einer Eingabe in einem Netzwerk oder das Einsammeln einer Ausgabe nicht betrachten. In Abschnitt 5.4.3 diskutieren wir, wer der Nutzer eines verteilten Eingabe-Ausgabe-Paars ist. In Abschnitt 5.4.4 stellen wir schließlich unsere Modellierung eines verteilten Eingabe-Ausgabe-Paars vor.

5.4.1 Eingabe-Ausgabe-Verhalten einer Komponente

Jede Komponente eines Netzwerks erhält für einen verteilten Algorithmus eine Teil der gesamten Eingabe. Der Teil der Eingabe einer Komponente ist die *Teileingabe* dieser Komponente. Die Teileingaben der Komponenten haben jeweils die gleiche Struktur, können sich jedoch in ihren Parametern unterscheiden. Zum Beispiel könnte jede Komponente eine Liste ihrer Nachbarn als Teileingabe erhalten. Je nach Position im Netzwerk sieht diese Liste dann für verschiedene Komponenten unterschiedlich aus. Jede Komponente führt startend von ihrer Teileingabe eine Berechnung aus und kommuniziert dabei gegebenenfalls mit anderen Komponenten im Netzwerk.

Der Teilalgorithmus einer Komponente berechnet die *Teilausgabe* der Komponente. Die Ausgabe eines verteilten Algorithmus besteht aus den Teilausgaben aller Komponenten und liegt entsprechend verteilt in einem Netzwerk vor. Die Abbildung 5.1 zeigt die Ausführung eines verteilten Algorithmus aus der Sicht einer Komponente eines Netzwerks.

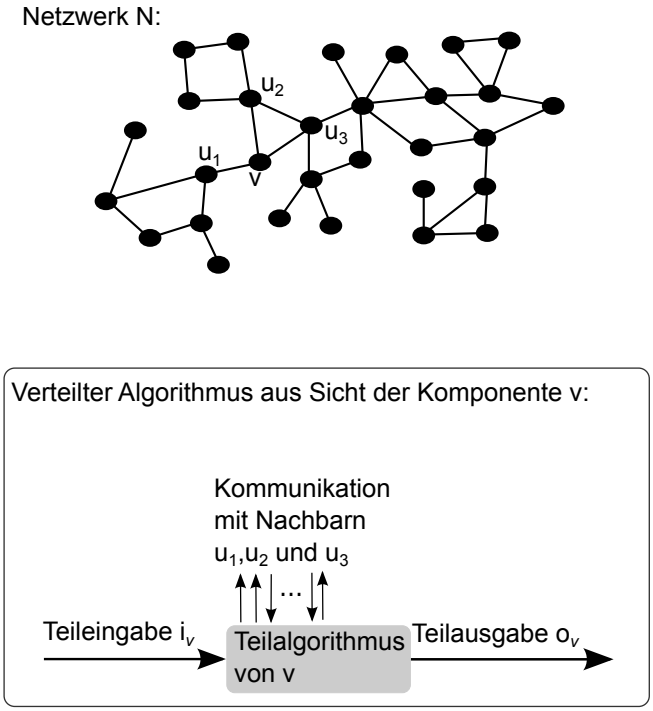


Abbildung 5.1: Im oberen Teil des Bildes ist ein Netzwerk N mit der Komponente v und ihren Nachbarn u_1, u_2 und u_3 zu sehen. Im unteren Teil des Bildes ist die Ausführung eines verteilten Algorithmus aus der Sicht der Komponente v dargestellt. Die Kommunikation der Komponente v mit ihren Nachbarn während der Ausführung des Teilalgorithmus ist durch ein- und ausgehende Pfeile dargestellt. Diese können einem beliebigen Muster folgen.

5.4.2 Verteilung einer Eingabe & Einsammeln einer Ausgabe

Wir beschäftigen uns weder mit einer Verteilung einer Eingabe, noch mit einem Einsammeln einer Ausgabe. Dies ist für die Betrachtung verteilter Algorithmen aus zwei Gründen üblich.

Der erste Grund ist, dass häufig weder das Verteilen der Eingabe noch das Einsammeln der Ausgabe nötig ist. Wir veranschaulichen diesen Umstand am Beispiel eines Leader-Election-Algorithmus [GRS18]. Das Netzwerk ist so initialisiert, dass jede Komponente ihre Nachbarn kennt. Diese Initialisierung dient als Eingabe des Algorithmus. Nach der Ausführung weiß genau eine Komponente, dass sie der Koordinator ist und jede andere Komponente weiß, dass sie es nicht ist. Diese Variante der Leader-Election wird auch implizite Variante genannt, weil alle Komponenten mit Ausnahme des Koordinators die Identität des Koordinators nicht kennen. Die verteilte Ausgabe eines impliziten Leader-Election-Algorithmus ist der gewählte Koordinator. Die Komponenten haben jedoch nicht das gleiche Wissen über die verteilte Ausgabe. Die so verteilt vorliegende Ausgabe könnte die Eingabe eines weiteren verteilten Algorithmus sein.

Der zweite Grund ist, dass das Verteilen der Eingabe und das Einsammeln der Ausgabe immer nach dem selben Schema erfolgen können. Zum Beispiel über die Berechnung eines Spannbaums mit einer Sammlung und Verteilung entlang des Baumes koordiniert durch die Wurzel. Die Verteilung der Eingabe und das Einsammeln der Ausgabe sind daher für die Betrachtung eines verteilten Algorithmus nicht sonderlich interessant.

5.4.3 Nutzer eines verteilten Eingabe-Ausgabe-Paars

Für einen sequentiellen Algorithmus nehmen wir einen Nutzer des Algorithmus an, der auch ein weiterer Algorithmus sein kann, der die Ausgabe als Eingabe erhält. Die Frage danach, wie ein Nutzer eines verteilten Algorithmus aussieht, ist interessant für die Laufzeitverifikation, da das Eingabe-Ausgabe-Paar für einen Nutzer verifiziert wird.

Analog zu einem sequentiellen Algorithmus, kann die verteilte Ausgabe auch eine verteilte Eingabe eines weiteren verteilten Algorithmus werden. In diesem Fall verwendet jede Komponente ihre Teilausgabe als neue Teileingabe. Die Komponenten bilden in diesem Fall den Nutzer, einen *verteilten Nutzer*. Wir haben darüber hinaus am Beispiel der impliziten Variante der Leader-Election gesehen, dass es nicht immer nötig ist, dass jede Komponente im Netzwerk die verteilte Ausgabe kennt.

Der Nutzer kann aber auch *zentralisiert* und auf eine zentralisierte Sammlung des Paares angewiesen sein. Nur wenn wir beide Fälle eines Nutzers bedenken, passt die Laufzeitverifikation zu einem verteilten System.

5.4.4 Modellierung eines verteilten Eingabe-Ausgabe-Paars

Wir befassen uns in diesem Abschnitt mit der Modellierung eines verteilten Eingabe-Ausgabe-Paars.

5.4.4.1 *Teileingabe und Teilausgabe*

Die Teileingabe und Teilausgabe einer Komponente entsprechen jeweils einer Belegung von Variablen mit Werten. Wir definieren deswegen eine Belegung.

Definition 7 (Belegung)**.** *Sei* A *eine Menge von Variablen und* B *eine Menge von Werten. Dann ist eine Funktion* $f : A \to B$ *eine* Belegung *von* A *mit* B*.*

Notation (Menge aller Belegungen)**.** *Wir bezeichnen die Menge aller Belegungen von* A *mit* B *als* $[A]$*, wenn die Menge der Werte* B *aus dem Kontext erkenntlich ist.*

Notation. *Für eine Belegung* f *schreiben wir anstelle von* $f(a) = b$ *verkürzt auch* $a = b$*.*

Die Begriffe Teileingabe und Teilausgabe definieren wir wie folgt.

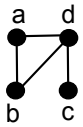
Definition 8 (Teileingabe, Teilausgabe)**.** *Sei* $N = (V, E)$ *ein Netzwerk. Seien* I *und* O *endliche Mengen an Variablen und seien* $\mathtt{Val}_I$ *und* $\mathtt{Val}_O$ *Mengen an Werten. Für jede Komponente* $v \in V$ *seien* $I_v \subseteq I$ *und* $O_v \subseteq O$ *Teilmengen an Variablen, sodass gilt* $I = \cup_{v \in V} I_v$ *und* $O = \cup_{v \in V} O_v$*.*

(i) Die Mengen $[I_v]$ *und* $[O_v]$ *aller Belegungen mit jeweils den Wertemengen* $\mathtt{Val}_I$ *und* $\mathtt{Val}_O$ *sind die Mengen der* Teileingaben *und* Teilausgaben *einer Komponente* $v \in V$*.*

(ii) Die Belegungen $i_v \in [I_v]$ *und* $o_v \in [O_v]$ *sind jeweils* Teileingabe *und* Teilausgabe *einer Komponente* $v \in V$*.*

Beispiel 1 (Leader-Election)**.** *Zur Illustration betrachten wir das Beispiel eines Leader-Election-Algorithmus in der Variante, in der jede Komponente die Identität des Koordinators kennt. Der Algorithmus wird auf dem, in der Abbildung 5.2 dargestellten, Netzwerk* N *mit den vier Komponenten* a*,* b*,* c *und* d *ausgeführt. Die Teileingabe einer Komponente ist ihre ID, sowie die IDs ihrer Nachbarn. Die Teilausgabe einer Komponente ist die ID des gewählten Koordinators. In diesem konkreten Fall haben die Komponenten die Komponente* b *als eindeutigen Koordinator gewählt. Jede Komponente hat daher als Teilausgabe* $\mathtt{leader} = b$*.*

Netzwerk:
N = (V,E):



Variablen:
ID = $\{id_v | v \in V\}$
NBRS = $\{nbrs_v | v \in V\}$
I = ID ∪ NBRS
O = {leader}
Für alle v ∈ V:
$I_v = \{id_v\} \cup \{nbrs_v\}$
O_v= {leader}

Werte:
$Val_I = V \cup P(V)$
$Val_O = V$

Signatur der Eingaben:
Für i ∈ [[I]] gilt
$i \subseteq i_{ID} \cup i_{NBRS}$
$i_{ID} \subseteq ID \times V$
$i_{NBRS} \subseteq NBRS \times P(V)$

Für alle v ∈ V:
Für $i_v \in [I_v]$ gilt
$i_v \subseteq i_{ID,v} \cup i_{NBRS,v}$
$i_{ID,v}$: $\{id_v\} \rightarrow V$
$i_{NBRS,v} : \{nbrs_v\} \rightarrow P(V)$

Teilein- und Teilausgabe der Komponente...
...a:
$i_a = \{(id_a, a), (nbrs_a, \{b, d\})\}$
$o_a = \{(leader, b)\}$
...b:
$i_b = \{(id_b, b), (nbrs_b, \{a, d\})\}$
$o_b = \{(leader, b)\}$
...c:
$i_c = \{(id_c, c), (nbrs_c, \{d\})\}$
$o_c = \{(leader, b)\}$
...d:
$i_d = \{(id_d, d), (nbrs_d, \{a, b, c\})\}$
$o_d = \{(leader, b)\}$

Abbildung 5.2: Teileingabe und Teilausgabe am Beispiel eines Leader-Election-Algorithmus. In diesem Fall ist die Komponente b der eindeutig gewählte Koordinator.

In der Abbildung 5.2 sehen wir, dass die Teileingabe i_v eine Belegung ist, die selbst aus zwei Belegungen, $i_{ID,v}$ und $i_{NBRS,v}$, zusammengesetzt ist. Die Belegung $i_{ID,v}$ ordnet der Variable für die ID einer Komponente einen entsprechenden Wert aus V zu. In künftigen Beispielen lassen wir die Angabe, wie sich die Belegung einer Teileingabe oder Teilausgabe zusammensetzt, weg; dies ist aus dem Kontext ersichtlich.

Wenn für zwei Komponenten gilt, dass ihre Teileingaben die gleiche Variable enthalten, dann teilen sich die Teileingaben eine Information. Gleiches gilt für Teilausgaben. Zum Beispiel teilen sich die Teilausgaben in dem Beispiel der Leader-Election die Information über den Koordinator in Form der Variable 'leader'.

Teileingabe und Teilausgabe sind in der Definition 8 jeweils vertauschbar, denn wir machen keine Annahmen über das Verhältnis ihrer Variablen- und Wertemengen zueinander. Wir definieren dennoch aus

Gründen der Lesbarkeit beide als Objekte. Für die verteilte Eingabe und Ausgabe eines verteilten Algorithmus verfahren wir genauso.

5.4.4.2 *Verteilte Eingabe und verteilte Ausgabe*

Eine verteilte Eingabe ist die Vereinigung der Teileingaben der Komponenten. Während bei einer Teileingabe jede Variable mit genau einem Wert belegt ist, kann bei einer verteilten Eingabe eine Variable verschiedene Belegungen haben. Gleiches gilt für die verteilte Ausgabe. Der Grund dafür ist, dass die Komponenten unabhängig von einander lokal rechnen. Wir führen eine *schwache* Belegung mit einer Relation anstelle einer Funktion ein.

Definition 9 (Schwache Belegung)**.** *Sei* A *eine Menge von Variablen und* B *eine Menge von Werten. Dann ist die Relation* $r \subseteq A \times B$ *eine* schwache Belegung *von* A *mit* B*.*

Notation (Menge aller schwachen Belegungen)**.** *Wir bezeichnen die Menge aller schwachen Belegungen von* A *mit* B *als* $[\![A]\!]$*, wenn* B *aus dem Kontext ersichtlich ist.*

Notation. *Für eine schwache Belegung* r *schreiben wir anstelle von* $(a, b) \in r$ *verkürzt auch* $a = b$*.*

Wir bezeichnen eine verteilte Eingabe und verteilte Ausgabe kurz als Eingabe und Ausgabe und definieren sie wie folgt.

Definition 10 (Eingabe, Ausgabe)**.** *Seien* $N = (V, E)$*,* I*,* O*,* Val_I *und* Val_O*, sowie für alle* $v \in V$ I_v*,* O_v*,* $[I_v]$ *und* $[O_v]$ *wie in Definition 8 (auf Seite 51) gegeben.*

(i) Die Mengen aller schwacher Belegungen $[\![I]\!] = \cup_{v \in V}[I_v]$ *und* $[\![O]\!] = \cup_{v \in V}[O_v]$ *sind die Mengen der* Eingaben *und* Ausgaben *in* N*.*

(ii) Sei eine Teileingabe $i_v \in [I_v]$ *für jedes* $v \in V$ *gegeben. Die schwache Belegung* $i = \cup_{v \in V} i_v$ *ist die zugehörige* Eingabe*. Analog für die* Ausgabe*.*

Beispiel 2 (Leader-Election)**.** *In dem Beispiel aus Abbildung 5.2 ist die Teileingabe einer Komponente jeweils ihre Nachbarschaft. Die Eingabe ist entsprechend das Netzwerk selbst, repräsentiert durch die überlappenden Nachbarschaften. Die Ausgabe* o *hat mit* $o = (\mathtt{leader}, b)$ *dieselbe Form wie eine der Teilausgaben, da alle Komponenten die gleiche Teilausgabe haben.*

5.4.4.3 *Teileingabe und Teilausgabe versus Eingabe und Ausgabe*

Wir modellieren ein Eingabe-Ausgabe-Paar so, dass bei einer Teileingabe (Teilausgabe) einer Komponente eine Variable nur mit einem einzigen Wert belegt sein kann, während bei der verteilten Eingabe (Ausgabe) eine Variable mit mehreren Werten belegt sein kann. Der

Grund für diese Modellierung ist, dass einerseits jede Komponente ihre Belegung unabhängig wählt und andererseits alle Komponenten ein gemeinsames Problem lösen.

Wir weisen daraufhin, dass es einfach ist bei einer Implementierung dafür zu sorgen, dass einer Variable lokal bei einer Komponente genau ein Wert zugewiesen wird; es reicht, einen entsprechenden Typ für die Variable zu wählen. Im Gegensatz dazu wäre ein zusätzlicher verteilter Algorithmus notwendig, um sicherzustellen, dass verschiedene Komponenten im Netzwerk die jeweilige Variable mit dem gleichen Wert belegen. So ein Algorithmus müsste dann einen Konsens bei der Variablenbelegung sicherstellen.

Beobachtung. *Wir können aus den Teileingaben aller Komponenten des Netzwerks auf die Eingabe schließen. Das folgt direkt aus der Definition 10. Andersherum können wir nicht von der Eingabe auf die einzelnen Teileingaben der Komponenten schließen.*

5.5 EINGABE-AUSGABE-SPEZIFIKATION

In diesem Abschnitt führen wir eine Spezifikation für Probleme ein, die durch verteilte Algorithmen gelöst werden sollen. Wir spezifizieren ein solches Problem durch eine *Vorbedingung* über Eingaben und eine *Nachbedingung* über Eingabe-Ausgabe-Paare. Damit bleiben wir also bei einer „klassische“ Methode, um Probleme für (sequentielle) Algorithmen zu spezifizieren.

In Abschnitt 5.5.1 führen wir zunächst lokale und globale Prädikate ein, mit denen wir jeweils den Zustand einer Komponente und den eines gesamten Netzwerks beschreiben. In Abschnitt 5.5.2 geben wir die Definition einer Eingabe-Ausgabe-Spezifikation an. In Abschnitt 5.5.3 diskutieren wir das Black-Box-Prinzip für unsere gewählte Eingabe-Ausgabe-Spezifikation.

5.5.1 Globale und lokale Prädikate

Im Gegensatz zu sequentiellen Algorithmen spezifizieren wir über ein *verteiltes* Eingabe-Ausgabe-Paar. Wir möchten deswegen sowohl eine Eigenschaft für das verteilte Eingabe-Ausgabe-Paar als auch für einzelne Komponenten ausdrücken. Dafür unterscheiden wir zwischen zwei Arten von Prädikaten in einem Netzwerk.

5.5.1.1 *Globale Prädikate*

Wir bezeichnen ein Prädikat als global in einem Netzwerk, wenn dessen Erfüllung von der verteilten Eingabe und/oder Ausgabe abhängt.

Definition 11 (Globales Prädikat). *Seien* $N = (V, E)$, I, O, Val_I, Val_O, $[\![I]\!]$ *und* $[\![O]\!]$ *wie in Definition 10 auf Seite 53 definiert. Sei* P *ein Prädikat, sodass* $P \subseteq [\![I]\!]$, $P \subseteq [\![O]\!]$ *oder* $P \subseteq [\![I]\!] \times [\![O]\!]$.

Das Prädikat P *ist* global *in* N.

Beispiel 3 (Leader-Election). *Als Beispiel für ein globales Prädikat betrachten wir ein Prädikat* P *über Eingaben. Die Eingabe sind hierbei Netzwerke.* P *ist erfüllt für ein Netzwerk, falls die Kanäle eines Netzwerks bidirektional sind.*

Wir betrachten das Netzwerk aus Abbildung 5.2. Das Prädikat P *ist mit der Eingabe* $i = i_a \cup i_b \cup i_c \cup i_d$ *erfüllt, denn für jedes* $v \in V$ *gilt: wenn* $u \in nbrs_v$, *dann auch* $v \in nbrs_u$.

5.5.1.2 *Lokale Prädikate*

Ein Prädikat ist *lokal* für eine Komponenten eines Netzwerks, wenn dessen Erfüllung ausschließlich von der Teileingabe und/oder Teilausgabe einer Komponente abhängt.

Definition 12 (Lokales Prädikat). *Seien* $N = (V, E)$, I, O, Val_I *und* Val_O, *sowie für alle* $v \in V$: I_v, O_v, $[I_v]$ *und* $[O_v]$ *wie in Definition 8 auf Seite 51 gegeben. Sei* $u \in V$ *und sei* p_u *ein Prädikat mit* $p_u \subseteq [I_u]$, $p_u \subseteq [O_u]$ *oder* $p_u \subseteq [I_u] \times [O_u]$,

Das Prädikat p_u *ist* lokal *für* u *in* N.

Beispiel 4 (Leader-Election). *Zur Illustration gehen wir erneut von dem Beispiel der Leader-Election aus, das in der Abbildung 5.2 dargestellt ist. Als Beispiel für ein lokales Prädikat für eine Komponente betrachten wir ein lokales Prädikat* $p_a \subseteq [I_a]$ *für die Komponente* a. *Das Prädikat* p_a *ist erfüllt, falls* a *nicht mit sich selbst benachbart ist, also* $id_a \notin nbrs_a$. *In dem konkreten Netzwerk aus Abbildung 5.2 ist das Prädikat* p_a *erfüllt:* $i_a \in p_a$, *denn* $id_a = a$ *und* $nbrs_a = \{b, d\}$; *folglich* $a \notin \{b, d\}$.

Wir können das Prädikat p_a auch generisch für alle Komponenten $v \in V$ definieren. Sei das Prädikat $p_v \subseteq [I_v]$ für eine Komponente $v \in V$ erfüllt, falls $id_v \notin nbrs_v$. Für alle $v \in V$ ist das Prädikate p_v jeweils lokal für v in N. Die Prädikate sind untereinander gleich bis auf positionsabhängige Parameter. In dem Netzwerk aus Abbildung 5.2 gilt das Prädikate p_v für die jeweilige Komponente v, denn alle Komponenten des Netzwerks besitzen die Eigenschaft, nicht Nachbar von sich selbst zu sein.

Im Falle einer Menge generisch definierter Prädikate, die lokal im Netzwerk sind, sprechen wir kurz von *einem* (generischen) lokalen Prädikat in einem Netzwerk.

5.5.1.3 *Globale Prädikate versus lokale Prädikate*

Globale und lokale Prädikate sind nicht einfach austauschbar:

Lemma 5.5.1. *Nicht jedes Prädikat, das global in einem Netzwerk ist, ist auch lokal in dem Netzwerk.*

Beweis. Das globale Prädikat P aus dem Beispiel 4 ist kein lokales Prädikat. Jede Komponente $v \in V$ besitzt zwar die Informationen über ihre eigenen Nachbarn $nbrs_v$ als Teil ihrer Teileingabe i_v. Eine Komponente hat jedoch keine Informationen darüber, wer die Nachbarn ihrer Nachbarn sind. Die Eigenschaft hängt also nicht nur von der Teileingabe *einer* Komponente ab. Damit ist das Prädikat P nicht lokal im Netzwerk. □

Die Unterscheidung zwischen lokalen und globalen Prädikaten ist in der Literatur verteilter Algorithmen nicht unüblich (vergleiche mit [Ray13, S. 166]). Ein lokales Prädikat ist dort jedoch über dem gesamten Zustand einer Komponente definiert und ein globales Prädikat über dem gesamten Zustand eines Netzwerks. Im Gegensatz dazu beschränken wir uns jeweils auf den beobachtbaren Teil des Zustands, also das Eingabe-Ausgabe-Verhalten einer Komponente oder eines Netzwerks.

Für die Eingabe-Ausgabe-Spezifikation benötigen wir nur globale Prädikate und noch keine lokalen Prädikate. Im folgenden Kapitel zu Zeugenprädikaten werden wir jedoch auf lokale Prädikate zurückkommen.

5.5.2 Definition: Eingabe-Ausgabe-Spezifikation

Wir spezifizieren das Eingabe-Ausgabe-Verhalten des verteilten Algorithmus und nicht seiner einzelnen Teilalgorithmen. Für die Spezifikation des Eingabe-Ausgabe-Verhaltens wählen wir deswegen globale Prädikate. Wir definieren eine Eingabe-Ausgabe-Spezifikation wie folgt:

Definition 13 (Eingabe-Ausgabe-Spezifikation)**.** *Seien* N, I, O, Val_I, Val_O, $[\![I]\!]$ *und* $[\![O]\!]$ *wie in der Definition 10 auf Seite 53.*

Seien $\phi \subseteq [\![I]\!]$ *und* $\psi \subseteq [\![I]\!] \times [\![O]\!]$ *globale Prädikate in* N.

(i) Das Tupel (ϕ, ψ) *ist eine* Eingabe-Ausgabe-Spezifikation *in* N.

(ii) Für korrekte Eingabe-Ausgabe-Paare (i, o) *gilt:*

$$\forall i \in [\![I]\!], o \in [\![O]\!] : \psi(i, o) \vee \neg\phi(i)$$

Beispiel 5 (Leader-Election)**.** *Beim Beispiel der Leader-Election setzt die Vorbedingung* ϕ *voraus, dass die Eingabe das tatsächliche Netzwerk präsentiert. Dafür müssen beispielsweise die Kanäle bidirektional sein, wofür wir*

ein globales Prädikat in Beispiel 3 angeben haben. Darüber hinaus müssen aber auch die IDs eindeutig sein. Für das Netzwerk aus der Abbildung 5.2 auf Seite 52 ist die Vorbedingung ϕ mit der Eingabe $i = i_a \cup i_b \cup i_c \cup i_d$ erfüllt.

Die Nachbedingung ψ besagt, dass der Koordinator eindeutig gewählt ist und der Koordinator eine Komponente des Netzwerks ist. Für das Netzwerk aus der Abbildung 5.2 ist die Nachbedingung ψ mit der Eingabe $i = i_a \cup i_b \cup i_c \cup i_d$ und der Ausgabe $o = o_a \cup o_b \cup o_c \cup o_d$ erfüllt.

In unserem Beispiel können wir nicht unterscheiden, ob nur eine Komponente den eindeutigen Koordinator gewählt hat oder alle Komponenten. Ist diese Eigenschaft gewünscht, könnte jede Komponente v eine mit ihrer ID indizierte Variable $\mathtt{leader}_v$ besitzt. Diese Lösung können wir generalisieren. Wir benötigen deswegen keine Vereinigung von Multimengen oder eine Zuordungsfunktion, die eine Ausgabe ihren jeweiligen Teilausgaben zuordnet.

5.5.3 Black-Box-Prinzip

In der Spezifikation sequentieller Algorithmen als Grundlagen für eine Zertifizierung kommt der Algorithmus selbst nicht vor. Dadurch dass nur über Eingaben und Ausgaben spezifiziert wird, wird ein *Black-Box-Prinzip* umgesetzt: Ein Zeugenprädikat und ein Checker können mit jedem zertifizierenden sequentiellen Algorithmus kombiniert werden, der passende Zeugen berechnet.

Wir haben mit der Eingabe-Ausgabe-Spezifikation das Black-Box-Prinzip auch für verteilte Algorithmen umgesetzt.

6 VERTEILBARE ZEUGENPRÄDIKATE

Wir führen in diesem Kapitel *Zeugenprädikate* für die Zertifizierung verteilter Algorithmen ein. Wir beginnen in Abschnitt 6.1 mit einem Beispiel, an dem wir die Konzepte, Zeuge und Zeugenprädikat, erst einmal intuitiv illustrieren.

Ein zertifizierender verteilter Algorithmus berechnet zusätzlich zur verteilten Ausgabe auch einen potenziellen Zeugen. In Abschnitt 6.2 erweitern wir das bekannte Eingabe-Ausgabe-Verhalten eines verteilten Algorithmus technisch um einen potenziellen Zeugen.

In Abschnitt 6.3 führen wir Zeugenprädikate für Eingabe-Ausgabe-Spezifikationen ein und fassen den Begriff des Zeugen formal.

In Abschnitt 6.4 diskutieren wir sequentielle Checker-Algorithmen zur Entscheidung eines Zeugenprädikats und erläutern, warum wir verteilte Checker-Algorithmen anstreben.

In Abschnitt 6.5 führen wir *verteilbare* Prädikate ein. Ein verteilbares Prädikat kann entschieden werden, indem einige lokale Prädikate je Komponente entschieden werden. Mit einem verteilbaren Zeugenprädikat verfolgen wir das Ziel eines verteilten Checker-Algorithmus.

6.1 BEISPIEL: ZEUGE FÜR EIN BIPARTITES NETZWERK

In diesem Abschnitt illustrieren wir die Konzepte, wie Zeuge und Zeugenprädikat, intuitiv. Als Beispiel wählen wir dafür einen verteilten Entscheidungsalgorithmus, der entscheidet, ob das Netzwerk, auf dem er läuft, bipartit ist [CH+16].

In Abschnitt 6.1.1 stellen wir den verteilten Bipartitheitstest vor. In Abschnitt 6.1.2 stellen wir die Konzepte der Zertifizierung im verteilten vor und in Abschnitt 6.1.3 illustrieren wir sie an einem konkreten Netzwerk.

6.1.1 Verteilter Bipartitheitstest

Wir definieren einen bipartiten Graphen:

Definition 14 (bipartiter Graph)**.** *Sei* $G = (V, E)$ *ein ungerichteter Graph.* G *ist genau dann* bipartit*, wenn nichtleere, disjunkte Teilmengen* $V_1 \subseteq V$ *und* $V_2 \subseteq V$ *existieren, sodass* $V = V_1 \cup V_2$ *und für alle* $e = (v_1, v_2) \in E$ *gilt* $v_1 \in V_1$ *und* $v_2 \in V_2$ *(oder* $v_2 \in V_1$ *und* $v_1 \in V_2$*).*

Eine *Bipartition* der Knotenmenge V sind zwei Partitionen V_1 und V_2 wie definiert.

Für den verteilten Bipartitheitstest entscheiden die Komponenten eines Netzwerks gemeinsam, ob das Netzwerk selbst ein bipartiter Graph ist. Hierbei kennt jede Komponente jeweils nur ihre eigene Nachbarschaft – die Teileingabe einer Komponente. Die verteilte Eingabe ist also das Netzwerk, auf dem der Bipartitheitstest ausgeführt wird.

Die Teilausgabe einer Komponente ist die Entscheidung 'yes' oder 'no'. Eine verteilte Ausgabe, die nur aus Teilausgaben 'yes' besteht, bedeutet, ein Netzwerk sei bipartit. Hingegen bedeutet eine verteilte Ausgabe, die aus Teilausgaben 'yes' und 'no' besteht, ein Netzwerk sei nicht bipartit. Wir verzichten an dieser Stelle darauf, den verteilten Algorithmus zu betrachten, da dies für die Illustration der Konzepte nicht nötig ist.

Wir nehmen die folgende Eingabe-Ausgabe-Spezifikation an. Vorbedingung für eine Eingabe ist, dass es sich tatsächlich um das Netzwerk handelt, auf dem der verteilte Bipartitheitstest läuft. Die Nachbedingung für ein Eingabe-Ausgabe-Paar ist, dass die Ausgabe genau dann bipartit heißt, wenn es sich auch um ein bipartites Netzwerk handelt.

Wir betrachten für unsere Illustration der Konzepte nur den „einfacheren" Fall eines bipartiten Netzwerks, stellen jedoch eine vollständige Fallstudie in Kapitel 9 vor. Einen Teil dieser Fallstudie haben wir in [Völ17] veröffentlicht.

6.1.2 Zertifizierung am Beispiel des Bipartitheitstests

Wir betrachten zunächst die Zertifizierung für einen sequentiellen und dann für einen verteilten Bipartitheitstest. Im Anschluss diskutieren wir die Unterschiede zwischen der Zertifizierung eines bipartiten Graphen und eines bipartiten Netzwerks.

6.1.2.1 *Zeuge für einen bipartiten Graphen*

Wir betrachten die Zertifizierung für den sequentiellen Bipartitheitstest. Ein Zeuge dafür, dass ein Graph bipartit ist, ist eine Bipartition des Graphen. Das Zeugenprädikat über dem Tripel bestehend aus Eingabe, Ausgabe und potenziellem Zeugen ist also erfüllt, wenn die Eingabe ein Graph G, die Ausgabe 'yes' und der Zeuge eine Bipartition von G ist.

Die Zeugeneigenschaft des Zeugenprädikats folgt unmittelbar aus der Definition 14 eines bipartiten Graphen. Die Abbildung 6.1 zeigt einen Graphen G, sowie eine Bipartition des Graphen G als Zeuge dafür, dass G bipartit ist.

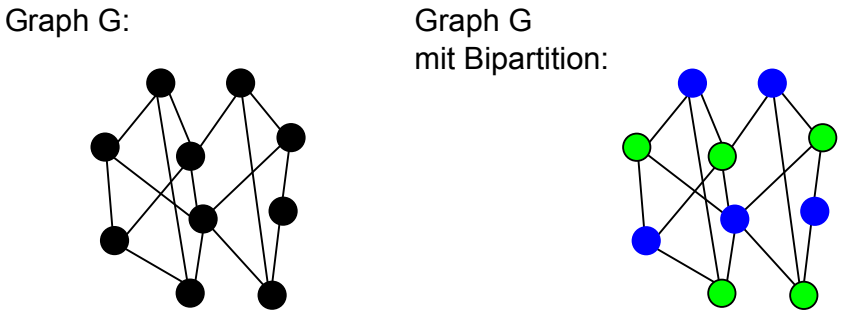


Abbildung 6.1: Die Abbildung zeigt auf der linken Seite den Graphen G und auf der rechten Seite eine Bipartition des Graphen G gekennzeichnet durch einerseits blaue und andererseits grüne Knoten mit schwarzer Umrandung. Die Bipartition ist ein Zeuge dafür, dass der Graph G bipartit ist.

6.1.2.2 *Verteilter Zeuge für ein bipartites Netzwerk*

Wir betrachten die Zertifizierung für den verteilten Bipartitheitstest. Wir übertragen den Zeugen, das Zeugenprädikat und die Zeugeneigenschaft des sequentiellen Bipartitheitstest auf den verteilten. Der verteilte Zeuge dafür, dass das Netzwerk bipartit ist, ist wieder eine Bipartition des Netzwerks.

Der Zeuge liegt jedoch verteilt als Teilzeugen der Komponenten im Netzwerk vor. Jede Komponente hat als Teilzeugen eine Bipartition ihrer Nachbarschaft. Das heißt jede Komponente hat eine andere Farbe als all ihre Nachbarn. Der Teilzeuge einer jeden Komponente enthält dabei sowohl ihre eigene Farbe als auch die der Nachbarn.

Die Teilzeugen benachbarter Komponenten teilen sich also Informationen. Sowohl der Teilzeuge einer Komponente a als auch die Teilzeugen all ihrer Nachbarn enthält eine Information zur Farbe der Komponente a. Technisch ausgedrückt bedeutet das, dass die Variablenmengen der Teilzeugen einen nicht-leeren Schnitt haben. Die Abbildung 6.2 illustriert die geteilten Informationen der Teilzeugen zweier benachbarter Komponenten a und b.

Das Zeugenprädikat ist ein globales Prädikat, das über Tripeln bestehend aus Eingabe, Ausgabe und Zeugen definiert ist. Das Zeugenprädikat ist erfüllt, wenn die Eingabe ein Netzwerk, die Ausgabe 'yes' und der Zeuge eine Bipartition des Netzwerks ist. Wie im sequentiellen Fall ergibt sich die Zeugeneigenschaft des Zeugenprädikats aus der Definition 14 eines bipartiten Graphen.

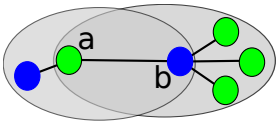


Abbildung 6.2: Zwei Komponenten a und b mit ihren Nachbarschaften und entsprechender Bipartition der Nachbarschaften. Die grauen Kreise zeigen welche Informationen der Teilzeugen der einer Komponente jeweils enthält. Die Überlappung der Kreise zeigt die geteilten Informationen der Teilzeugen.

Es gibt zusätzlich ein lokales Prädikat, das über Tripeln bestehend aus Teileingabe, Teilausgabe und Teilzeugen einer Komponente definiert ist. Das lokale Prädikat ist für eine Komponente erfüllt, wenn die Teileingabe die Nachbarschaft der Komponente, die Teilausgabe 'yes' und der Teilzeuge eine Bipartition der Nachbarschaft ist.

Das globale Zeugenprädikat und das lokale Prädikat hängen wie folgt miteinander zusammen: Wenn das lokale Prädikat für alle Komponenten des Netzwerks erfüllt ist, dann gilt das globale Zeugenprädikat im Netzwerk. Wir nennen diese Implikation die *Verteilungseigenschaft* des Zeugenprädikats. Die Verteilungseigenschaft gilt, denn die Bipartitionen der Nachbarschaften überlappen sich und bilden somit eine Bipartition des Netzwerks. Wenn ein lokales Prädikat für alle Komponenten gelten muss, bezeichnen wir das Zeugenprädikat als *universell-verteilbar* im Netzwerk.

Alle Teilzeugen sind gleich aufgebaut. Der Zeuge ist deswegen *gleichverteilt*. Jeder Teilzeuge einer Komponente enthält lediglich Informationen, die sich auf die Nachbarschaft der Komponente beschränken. Der Zeuge in diesem Beispiel ist deswegen *beschränkt*. Es gibt keinen Teilzeugen, der alle Informationen des Zeugen beinhaltet. Der Zeuge ist deswegen *nicht zentralisiert*.

Die geteilten Informationen der Teilzeugen benachbarter Komponenten sind unabdingbar für die Verteilungseigenschaft und somit für das Korrektheitsargument des verteilten Zeugen. Die Teilzeugen müssen dafür jedoch in allen geteilten Informationen miteinander übereinstimmen. Wir bezeichnen einen verteilten Zeugen, der in den geteilten Informationen seiner Teilzeugen widerspruchsfrei ist, als *konsistent*.

6.1.2.3 *Unterschiede: Zeuge im sequentiellen und im verteilten Fall*

Bei dem aufgezeigten Beispiel unterscheidet sich die Zertifizierung im verteilten Fall in einigen Punkten von der Zertifizierung im sequentiellen Fall. Das Zeugenprädikat weist im vorgestellten Beispiel die folgenden Unterschiede auf:

- Es gibt ein *globales* Zeugenprädikat (siehe Abschnitt 6.3).
- Das Zeugenprädikat ist *universell-verteilbar* (siehe Abschnitt 6.5).

- Es gibt ein *lokales* Prädikat für das Zeugenprädikat (siehe Abschnitt 6.5).

Wir beschäftigen uns in diesem Kapitel mit dem Zeugenprädikat im Allgemeinen. Dabei finden wir in den angegebenen Abschnitten die Unterschiede wieder, die wir hier am Beispiel bereits demonstriert haben. Wir führen in diesem Kapitel bereits formal den Begriff des Zeugen ein, behandeln Zeugen jedoch erst detaillierter im folgenden Kapitel 7.

Der vorgestellte Zeuge weist die folgenden Unterschiede zu dem Zeugen des zertifizierenden sequentiellen Algorithmus auf:

- Der Zeuge ist *gleichverteilt*, *beschränkt* und nicht *zentralisiert* (siehe Abschnitt 7.1).
- Der Zeuge besteht aus Teilzeugen, die sich Informationen teilen (siehe Abschnitt 7.2).
- Der Zeuge ist *konsistent* (siehe Abschnitt 7.3).

6.1.3 Illustration an einem konkreten Netzwerk

Wir illustrieren das Beispiel des verteilten Bipartitheitstest mit dem eingeführten verteilten Zeugen an einem konkreten Netzwerk. Die Abbildung 6.3 zeigt das Netzwerk N. Das Netzwerk N ist bipartit, gekennzeichnet in der Abbildung durch eine Bipartition mit schwarz und weiß gefärbten Komponenten.

Die Abbildung zeigt die Teileingabe, Teilausgabe und den Teilzeugen der Komponente 3 und der Komponente 6. Die Komponente 3 hat die Komponente 6 als einzigen Nachbarn. Entsprechend sehen wir in der Teileingabe i_3 der Komponente 3, dass für die Variable $\text{nbrs}_3 \in I_3$ gilt $\text{nbrs}_3 = \{6\}$. Die Teilausgabe der Komponente 3 ist die Entscheidung, dass das Netzwerk bipartit ist: bipartite $=$ yes. Der Teilzeuge der Komponente 3 ist die eigene Farbe und die Farbe der Nachbarn, in dem Fall nur der Komponente 6.

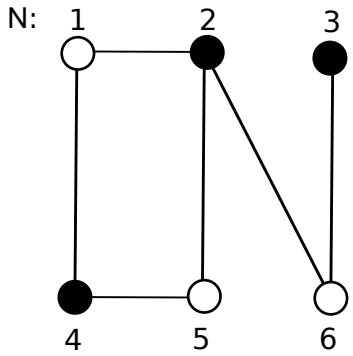
Die Komponente 6 hat zwei Nachbarn, Komponente 2 und Komponente 3. Entsprechend gilt für die Teileingabe der Komponente 6: $\text{nbrs}_6 = \{2, 3\}$. Die Teilausgabe der Komponente 6 ist wie bei der Komponente 3 die Entscheidung, dass das Netzwerk N bipartit ist. Der Teilzeuge der Komponente 6 enthält die eigene Farbe, sowie die Farben der Nachbarn, also der Komponente 2 und der Komponente 3.

Die Teilzeugen der beiden Komponenten besitzen geteilte Informationen, da sie benachbart sind. Beide besitzen ihre jeweilige Farbe, sowie die der anderen als Teil der Bipartition ihrer jeweiligen Nachbarschaft:

$$w_3 \cap w_6 = \{(\text{color}_3, \text{black}), (\text{color}_6, \text{white})\}$$

.

Netzwerk:
$N = (V, E)$



Wertemenge:
$Val_I = V \cup P(V)$
$Val_O = \{yes, no\}$
$Val_W = \{black, white\}$

Variablenmengen:
$I = \{id_v \mid v \in V\} \cup \{nbrs_v \mid v \in V\}$
$O = \{bipartite\}$
$W = \{color_v \mid v \in V\}$
Für alle $v \in V$:
$I_v = \{id_v\} \cup \{nbrs_v\}$
$O_v = \{bipartite\}$
$W_v = \{color_v\} \cup \{color_u \mid u \text{ ist Nachbar von } v\}$

Teileingabe, Teilausgabe und Teilzeuge der Komponente...
...3:
$i_3 = \{(id_3, 3), (nbrs_3, \{6\})\}$
$o_3 = \{(bipartite, yes)\}$
$w_3 = \{(color_3, black), (color_6, white)\}$

...6:
$i_6 = \{(id_6, 6), (nbrs_6, \{2, 3\})\}$
$o_6 = \{(bipartite, yes)\}$
$w_6 = \{(color_6, white), (color_2, black), (color_3, black)\}$

Abbildung 6.3: Beispiel eines bipartiten Netzwerks mit den Teileingaben, Teilausgaben und Teilzeugen der Komponente 3 und der Komponente 6 beim verteilten Bipartitheitstest. $P(V)$ bezeichnet die Potenzmenge von V.

6.2 EINGABE-AUSGABE-VERHALTEN ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Ein zertifizierender sequentieller Algorithmus berechnet zusätzlich zur ursprünglichen Ausgabe einen Zeugen. Genauso berechnet eine zertifizierende Variante eines verteilten Algorithmus zusätzlich einen Zeugen. Genauer gesagt handelt es sich bei dem berechneten Zeugen zunächst um einen *potenziellen* Zeugen, solange nicht bekannt ist, ob der Zeuge das Eingabe-Ausgabe-Paar des ursprünglichen Algorithmus

tatsächlich verifiziert. Wir erweitern in diesem Abschnitt die Ausgabe technisch um einen potenziellen Zeugen.

In Abschnitt 6.2.1 führen wir ein, wie die Ausgabe eines zertifizierenden verteilten Algorithmus aufgebaut ist. In Abschnitt 6.2.2 diskutieren wir, wie die erweitere Ausgabe mit einer Eingabe-Ausgabe-Spezifikation zusammenpasst. In Abschnitt 6.2.3 halten wir einige Objekte fest, die gemeinsam das Interface eines zertifizierenden verteilten Algorithmus bilden. Unser Ziel ist dabei, im Folgenden für eine bessere Lesbarkeit zu sorgen.

6.2.1 Ausgabe eines zertifizierenden verteilten Algorithmus

Ein potenzieller Zeuge ist Teil der Ausgabe einer zertifizierenden Variante eines verteilten Algorithmus. Damit gilt für einen potenziellen Zeugen alles, was auch für eine übliche Ausgabe gilt. Die Ausgabe eines zertifizierenden verteilten Algorithmus besteht dann aus zwei Teilen: der ursprünglichen Ausgabe eines verteilten Algorithmus und dem potenziellen Zeugen einer zertifizierenden Variante.

6.2.1.1 *Ursprüngliche Ausgabe*

Seien die ursprünglichen Ausgaben (siehe Definition 10) wie folgt gegeben:

- endliche Variablenmenge O,
- Wertemenge Val_O,
- Variablenmengen $O_v \subseteq O$ je Komponente $v \in V$,
- potenziellen Teilzeugen $[O_v]$ je Komponente $v \in V$, sowie
- potenzielle Zeugen $[\![O]\!]$.

6.2.1.2 *Potenzielle Zeugen*

Seien die potenziellen Zeugen (siehe ebenfalls Definition 10) zusätzlich wie folgt gegeben:

- endliche Variablenmenge W,
- Wertemenge Val_W,
- Variablenmengen $W_v \subseteq W$ je Komponente $v \in V$,
- potenziellen Teilzeugen $[W_v]$ je Komponente $v \in V$, sowie
- potenzielle Zeugen $[\![W]\!]$.

6.2.1.3 *Zusammengesetzte Ausgabe*

Die zusammengesetzte Ausgabe einer zertifizierenden Variante ist dann wie folgt gegeben:

- die neue Variablenmenge ist $O_{zert} = O \times W$,
- die neue Wertemenge ist $Val_{zert} = Val_O \times Val_W$,
- die neuen Variablenmengen je Komponente $v \in V$ sind $O_{zert,v} = O_v \times W_v$,
- die neuen Teilausgaben je Komponente $v \in V$ sind $[O_{zert,v}]$ je Komponente $v \in V$ und
- die neuen Ausgaben sind $[\![O_{zert}]\!]$.

Ein Eingabe-Ausgabe-Paar eines zertifizierenden verteilten Algorithmus hat also die Form $(i,(o,w))$ mit der Eingabe i und der Ausgabe (o,w), wiederum bestehend aus der ursprünglichen Ausgabe o und dem potenziellen Zeugen w. Wir bezeichnen jedoch auch bei einer zertifizierenden Variante nur die ursprüngliche Ausgabe o weiterhin als Ausgabe. Sprechen wir also bei einem zertifizierenden verteilten Algorithmus von einem Eingabe-Ausgabe-Paar, so beziehen wir uns auf das Eingabe-Ausgabe-Paare des ursprünglichen verteilten Algorithmus.

6.2.2 Eingabe-Ausgabe-Spezifikation

Der potenzielle Zeuge ist Teil des beobachtbaren Zustands eines Netzwerks. Für ein globales Prädikat (siehe Definition 11) bedeutet das, dass es auch zusätzlich oder alleinig über einem potenziellen Zeugen definiert sein kann. Genauso bedeutet das für ein lokales Prädikat (siehe Definition 12), dass es auch zusätzlich oder alleinig über einem Teilzeugen definiert sein kann.

Wenn wir von der Eingabe-Ausgabe-Spezifikation im Kontext einer zertifizierenden Variante eines verteilten Algorithmus sprechen, dann beziehen wir uns auf die Eingabe-Ausgabe-Spezifikation des ursprünglichen verteilten Algorithmus. Weiterhin schreiben wir für ein Eingabe-Ausgabe-Paar $(i,(o,w))$ eines zertifizierenden verteilten Algorithmus zur Vereinfachung das Tripel (i,o,w).

6.2.3 Interface

Wir nehmen für Definitionen, Theoreme und Beweise dieses Kapitels immer wieder ein Netzwerk, Eingaben, Ausgaben, potenzielle Zeugen, Teileingaben, Teilausgaben, Teilzeugen, sowie eine Eingabe-Ausgabe-

Objekt	Bedeutung
$N = (V, E)$	Netzwerk
I	Variablen für Eingaben
O	Variablen für Ausgaben
W	Variablen für potenzielle Zeugen
Val_I	Werte für Eingaben
Val_O	Werte für Ausgaben
Val_W	Werte für potenzielle Zeugen
$[\![I]\!]$	Eingaben
$[\![O]\!]$	Ausgaben
$[\![W]\!]$	potenzielle Zeugen
I_v	Variablen für Teileingaben je Komponente $v \in V$
O_v	Variablen für Teilausgaben je Komponente $v \in V$
W_v	Variablen für Teilzeugen je Komponente $v \in V$
$[I_v]$	Teileingaben je Komponente $v \in V$
$[O_v]$	Teilausgaben je Komponente $v \in V$
$[W_v]$	Teilzeugen je Komponente $v \in V$
(ϕ, ψ)	Eingabe-Ausgabe-Spezifikation

Abbildung 6.4: Auflistung der Objekte, auf die wir künftig referenzieren. Es sind zum eine beliebiges Netzwerk und eine beliebige Eingabe-Ausgabe-Spezifikation aufgelistet und zum anderen alle Objekte, die gemeinsam das Interface eines zertifizierenden verteilten Algorithmus bilden.

Spezifikation an. All diese Objekte bilden gemeinsam das Interface eines zertifizierenden verteilten Algorithmus.

Für eine bessere Lesbarkeit nehmen wir die entsprechenden Objekte, wie in Abbildung 6.4 aufgelistet als beliebig, aber fest an. Wir führen die Bedeutung der Objekte in der Tabelle zum einfachen Nachschlagen mit auf. Für ein detailliertes Nachschlagen verweisen wir auf die folgenden Definitionen: Definition 6 (siehe Seite 41), Definition 8 (siehe Seite 51), Definition 10 (siehe Seite 53) und Definition 13 (siehe Seite 56).

In Definitionen, Theoremen und Beweisen führen wir die Objekte im Folgenden lediglich auf, schreiben dazu *„wie üblich definiert““* und referenzieren auf diesen Abschnitt. Geben wir konkrete Objekte an, wie in Beispielen, dann machen wir das durch den Kontext erkenntlich.

6.3 ZEUGENPRÄDIKATE

In diesem Abschnitt führen wir Zeugenprädikate für Eingabe-Ausgabe-Spezifikationen ein. In Abschnitt 6.3.1 definieren wir Zeugenprädikate und in Abschnitt 6.3.2 Zeugen. In Abschnitt 6.3.3 diskutieren wir die Unterschiede der beiden eng verwobenen Konzepte des Zeugenprädikats und des Zeugen. Wir führen in Abschnitt 6.3.4 außerdem *vollständige* Zeugenprädikate ein.

6.3.1 Definition eines Zeugenprädikats

Wir definieren ein Zeugenprädikat für eine Eingabe-Ausgabe-Spezifikation über den Eingaben, Ausgaben und potenziellen Zeugen eines zertifizierenden verteilten Algorithmus:

Definition 15 (Zeugenprädikat, Zeugeneigenschaft)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ) *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Sei $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein globales Prädikat in* N *mit der folgenden Eigenschaft:*

$$\forall i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!] : \quad \Gamma(i, o, w) \longrightarrow (\psi(i, o) \vee \neg\phi(i)) \tag{6.1}$$

(i) Γ *ist ein* Zeugenprädikat *für* (ϕ, ψ).

(ii) Die Eigenschaft (6.1) *des Zeugenprädikats ist die* Zeugeneigenschaft *für* (ϕ, ψ).

Beispiel 6 (Bipartites Netzwerk)**.** *Wir erinnern uns an das Beispiel des verteilten Bipartitheitstest (siehe Abschnitt 6.1. Das Zeugenprädikat ist über dem Tripel bestehend aus Eingabe, Ausgabe und potenziellen Zeugen definiert; es ist erfüllt, wenn die Eingabe ein Netzwerk, die Ausgabe 'yes' (bipartit) und der potenzielle Zeuge eine Bipartition des Netzwerks ist.*

Die Zeugeneigenschaft des Zeugenprädikats folgt unmittelbar aus der Definition 14 eines bipartiten Graphen auf Seite 59. Eine Bipartition der Komponenten eines Netzwerks impliziert, dass das Netzwerk bipartit ist. Ist das Zeugenprädikat erfüllt, so ist das konkrete Eingabe-Ausgabe-Paar folglich korrekt.

Der verteilte Algorithmus selbst kommt in der Definition 15 des Zeugenprädikats nicht vor. Das Black-Box-Prinzip zertifizierender sequentieller Algorithmen bleibt also nicht nur bei der Eingabe-Ausgabe-Spezifikation, sondern auch beim Zeugenprädikat erhalten. Folglich kann ein Zeugenprädikat für jeden zertifizierenden verteilten Algorithmus mit entsprechender Eingabe-Ausgabe-Spezifikation genutzt werden kann. Eine zertifizierende Variante eines verteilten Algorithmus muss dann einen zum Zeugenprädikat passenden Zeugen berechnen.

6.3.2 Definition eines Zeugen

Wir fassen den Begriff des Zeugen formal:

Definition 16 (Zeuge). *Seien* N, I, O, W, $\mathtt{Val}_I$, $\mathtt{Val}_O$, $\mathtt{Val}_W$, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ) *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein Zeugenprädikat für* (ϕ, ψ).

$w \in [\![W]\!]$ *ist ein* Zeuge *für die Korrektheit eines Eingabe-Ausgabe-Paars* $(i, o) \in [\![I]\!] \times [\![O]\!]$ *bezüglich* (ϕ, ψ), *falls* $(i, o, w) \in \Gamma$.

Beispiel 7 (Bipartites Netzwerk). *In dem einführenden Beispiel, dargestellt in der Abbildung 6.3 auf Seite 64, sind nur die Teilzeugen der Komponente 3 und der Komponente 6 aufgezeigt. Anhand der grafischen Darstellung des Netzwerks, können wir den potenziellen Zeugen w dennoch wie folgt angeben:*

$$\begin{aligned} w = \{&(\text{color}_1, \text{white}),\\ &(\text{color}_2, \text{black}),\\ &(\text{color}_3, \text{black}),\\ &(\text{color}_4, \text{black}),\\ &(\text{color}_5, \text{white}),\\ &(\text{color}_6, \text{white})\} \end{aligned}$$

Der potenzielle Zeuge w ist ein Zeuge für das Eingabe-Ausgabe-Paar, bei dem die Eingabe das (bipartite) Netzwerk aus Abbildung 6.3 ist und die Ausgabe bipartit lautet. Denn mit diesem Eingabe-Ausgabe-Paar erfüllt der Zeuge w das Zeugenprädikat aus dem Beispiel 6. Der Zeuge ist eine Bipartition und verifiziert, dass das Netzwerk bipartit ist.

Ist das Zeugenprädikat also erfüllt mit einem Tripel (i, o, w) bestehend aus Eingabe i, Ausgabe o und potenziellem Zeugen w, so ist der potenzielle Zeuge w ein Zeuge für das Eingabe-Ausgabe-Paar (i, o). Wir unterscheiden im Weiteren nicht explizit zwischen einem potenziellen Zeugen und einem Zeugen, wenn sich diese Unterscheidung durch den Kontext ergibt.

6.3.3 Zeuge versus Zeugenprädikat

Bei der Laufzeitverifikation haben wir zwei unterschiedliche Anforderungen, die sich jeweils in den Konzepten, Zeuge und Zeugenprädikat, widerspiegeln.

Die eine Anforderung ist, dass ein zertifizierender verteilter Algorithmus *ein* Eingabe-Ausgabe-Paar zur Laufzeit verifiziert und das unabhängig von anderen Eingabe-Ausgabe-Paaren des Algorithmus. Diese Anforderung bedient ein zertifizierender verteilter Algorithmus, indem er einen Zeugen für die Korrektheit eines Eingabe-Ausgabe-

Paar berechnet. Dadurch können wir einen verteilten Algorithmus auch dann erfolgreich benutzen, wenn er nicht immer die korrekte Ausgabe zu einer Eingabe berechnet. Sei es weil der Algorithmus fehlerhaft ist oder weil er auf Heuristiken basiert.

Die andere Anforderung ist, dass ein korrekter zertifizierender verteilter Algorithmus *jedes* seiner Eingabe-Ausgabe-Paare zur Laufzeit verifiziert. Diese Anforderung wird mit einem Zeugenprädikat bedient; ein Zeugenprädikat gibt ein Schema für die Art der Zeugen vor. Intuitiv gesprochen ist ein Zeuge ein Korrektheitsargument und ein Zeugenprädikat liefert die Argumentationsstruktur für die Korrektheitsargumente.

Während wir einen Zeugen für *eine Probleminstanz* – also ein Eingabe-Ausgabe-Paar – definieren, definieren wir ein Zeugenprädikat für *ein Problem* – also alle Eingabe-Ausgabe-Paare. Diese beiden Anforderungen an die Laufzeitverifikation spiegeln sich also jeweils in dem Konzept eines Zeugen und eines Zeugenprädikats wider.

6.3.4 Vollständige Zeugenprädikate

Für ein Zeugenprädikat gilt:

Lemma 6.3.1. *Ein globales Prädikat über Eingabe, Ausgabe und potenziellem Zeugen, das nie erfüllt ist, ist ein Zeugenprädikat für eine beliebige Eingabe-Ausgabe-Spezifikation.*

Beweis. Folgt aus der Definition 15 direkt aus der Zeugeneigenschaft (6.1) eines Zeugenprädikats. □

Solch ein Zeugenprädikat ist nicht hilfreich, um das Problem der Instanzverifikation zu lösen. Entsprechend ist es auch nicht erstrebenswert.

Wir definieren ein Zeugenprädikat dennoch so, um der Perspektive eines Nutzers eines zertifizierenden verteilten Algorithmus gerecht zu werden. Für den Nutzer ist es ausreichend, wenn das Zeugenprädikat mit seinem konkreten Eingabe-Ausgabe-Paar und Zeugen erfüllt ist. Ob es zu jedem korrekten Eingabe-Ausgabe-Paar einen entsprechenden Zeugen gibt, ist für ihn irrelevant.

Eine Entwicklerin eines zertifizierenden verteilten Algorithmus hingegen muss sicherstellen, dass ihr Algorithmus korrekt ist. Ein korrekter zertifizierender verteilter Algorithmus berechnet korrekte Eingabe-Ausgabe-Paare und zusätzlich einen Zeugen für das Paar. Ein entsprechendes Zeugenprädikat bezeichnen wir als *vollständig*:

Definition 17 (Vollständiges Zeugenprädikat)**.** *Seien* N, I, O, W, $\mathtt{Val}_I$, $\mathtt{Val}_O$, $\mathtt{Val}_W$, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ) *wie üblich definiert (siehe Ab-*

schnitt 6.2.3 auf Seite 66). Sei $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein Zeugenprädikat für* (ϕ, ψ).

Γ *ist* vollständig, *falls*

$$\forall i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!] : (\psi(i, o) \vee \neg\phi(i)) \longrightarrow \Gamma(i, o, w)$$

Das Zeugenprädikat aus dem Beispiel 6 des verteilten Bipartitheitstest ist nur für den Fall der Entscheidung eines bipartiten Netzwerks vollständig. Wir erweitern das Zeugenprädikat um den Fall der Entscheidung eines nicht bipartiten Netzwerks in der Fallstudie zum Bipartitheitstest in Kapitel 9 und betrachten hier nur den Fall eines bipartiten Netzwerks.

Beispiel 8 (Bipartites Netzwerk)**.** *Wir betrachten alle korrekten Eingabe-Ausgabe-Paare, bei denen die Ausgabe die Entscheidung ist, das Netzwerk ist bipartit. Für jedes solche Paar gibt es per Definition eine Bipartition – also einen Zeugen, mit dem das Zeugenprädikat erfüllt ist. Das Zeugenprädikat ist folglich vollständig.*

Lemma 6.3.2 (Existenz eines Zeugen)**.** *Wenn das Zeugenprädikat einer Eingabe-Ausgabe-Spezifikation vollständig ist, dann existiert für jedes korrekte Eingabe-Ausgabe-Paar ein Zeuge.*

Beweis. Folgt aus der Definition 17 eines vollständigen Zeugenprädikats. □

Eine Entwicklerin eines zertifizierenden verteilten Algorithmus muss also für die Korrektheit ihres Algorithmus unter anderem zeigen, dass ihr gewähltes Zeugenprädikat vollständig ist. Ein Nutzer des Algorithmus hingegen ist lediglich an der Korrektheit eines Eingabe-Ausgabe-Paares interessiert. Für ihn reicht es, wenn es genau für dieses Paar einen Zeugen gibt mit dem das Zeugenprädikat erfüllt ist. Für den Nutzer ist es also nur notwendig von der Zeugeneigenschaft eines Zeugenprädikats überzeugt zu sein. Die Vollständigkeit eines Zeugenprädikats ist grundlegend für die Korrektheit eines zertifizierenden verteilten Algorithmus, nicht aber für die Instanzverifikation zur Laufzeit.

6.4 SEQUENTIELLE CHECKER

Eine direkte Variante das Konzept eines zertifizierenden sequentiellen Algorithmus auf terminierende verteilte Algorithmen zu übertragen ist die Einführung von Checkern, die ein Zeugenprädikat sequentiell entscheiden.

In Abschnitt 6.4.1 skizzieren wir die Arbeitsweise eines sequentiellen Checkers. In Abschnitt 6.4.2 erläutern wir die Nachteile eines sequentiellen Checkers, wobei wir insbesondere auf die Kerngedanken verteilter Algorithmen eingehen: Verteiltheit, Gleichheit und Lokalität. In Abschnitt 6.4.3 formulieren wir unser Ziel nach verteilten Checkern und stellen vorgreifend die angestrebte Checker-Architektur vor als Motivation für verteilbare Zeugenprädikate. Verteilte Checker betrachten wir dann jedoch erst in Teil iv.

6.4.1 Arbeitsweise eines sequentiellen Checkers

Die Arbeitsweise eines sequentiellen Checkers sieht wie folgt aus. Nach der festgestellten Terminierung (siehe Kapitel 5, Abschnitt 5.3) sendet jede Komponente des Netzwerks dem sequentiellen Checker ihre Teileingabe, Teilausgabe und den Teilzeugen. Der Checker berechnet aus den Teileingaben, Teilausgaben und Teilzeugen aller Komponenten die Eingabe, Ausgabe und den Zeugen. Für dieses Tripel entscheidet der Checker das Zeugenprädikat dann mit einer sequentiellen Entscheidungsprozedur.

6.4.1.1 *Zentralität eines sequentiellen Checkers*

Ein sequentieller Checker muss auf einer Komponente ausgeführt werden, die mit allen Komponenten des Netzwerks benachbart ist. Der Grund dafür ist, dass Komponenten nicht vertrauenswürdig sind (siehe Abschnitt 5.2); Wir bezeichnen eine Komponente in einem Netzwerk, die mit allen anderen Komponenten eines Netzwerks benachbart ist, als *zentral*.

Theorem 6.4.1 (Zentralität eines sequentiellen Checkers)**.** *Eine sequentielle Checker-Komponente muss für die Verifikation eines verteilten Eingabe-Ausgabe-Paars in einem Netzwerk zentral in dem Netzwerk sein.*

Beweis. Nehmen wir an, eine Komponente sei nicht mit dem Checker benachbart. Diese Komponente sendet ihre Teileingabe, ihre Teilausgabe und ihren Teilzeugen über andere Komponenten an den Checker. Das Tripel kommt beim Checker an, sofern mindestens eine Komponente mit dem Checker benachbart ist und alle Komponenten auf dem Weg die Nachricht weitersenden.

Wenn jedoch eine Komponente auf dem Weg die Nachricht kompromittiert und ohne Beschränkung der Allgemeinheit die Teilausgabe verändert, dann berechnet der Checker auf dieser Grundlage eine Ausgabe, die so nicht verteilt im Netzwerk vorliegt. Verifiziert der sequentielle Checker das von ihm so berechnete Eingabe-Ausgabe-Paar, handelt es sich folglich nicht um das berechnete Paar. □

Gehen wir von vertrauenswürdigen Komponenten aus, so genügt es, wenn eine sequentielle Checker-Komponente mit mindestens einer Komponente eines Netzwerks benachbart ist. In diesem Fall würden wir jedoch vom Fehlermodell zertifizierender Algorithmen abweichen, da wir eine Komponente nicht mehr als Black-Box betrachten würden. Des Weiteren gäbe es keine Notwendigkeit mehr das Eingabe-Ausgabe-Paar zu verifizieren, da alle Komponenten bereits vertrauenswürdig arbeiten.

6.4.1.2 *Zentrale sequentielle Checker*

Im Falle eines sequentiellen Checkers sind bei einem zertifizierenden verteilten Algorithmus im Vergleich zu einem zertifizierenden sequentiellen Algorithmus die folgenden zusätzlichen Schritte notwendig:

- Der Zeuge wird verteilt von den Komponenten des Netzwerks berechnet.
- Jede Komponente sendet ihre Teileingabe, Teilausgabe und ihren Teilzeugen an den zentralen sequentiellen Checker.
- Der zentrale sequentielle Checker berechnet aus allen Teileingaben, Teilausgaben und Teilzeugen jeweils die Eingabe, Ausgabe und den Zeugen.

Mit der Einführung eines zentralen sequentiellen Checkers gelingt es uns, das Konzept eines zertifizierenden sequentiellen Algorithmus so auf verteilte Algorithmen zu übertragen, dass es nah am ursprünglichen Konzept ist.

Ein zentraler sequentieller Checker ist in der Laufzeitverifikation verteilter Systeme keineswegs unüblich. Er ist sogar eher die Regel und verteilte Ansätze zur Laufzeitverifikation gibt es im Vergleich wenige; die Entwicklung solcher Ansätze ist deswegen auch ein aktives Forschungsgebiet [MB15].

6.4.2 Nachteile sequentieller Checker

Es gibt einige Nachteile eines sequentiellen Checkers. Zwar können wir unser Ziel, nah am ursprünglichen Konzept eines zertifizierenden Algorithmus zu bleiben, mit einem sequentiellen Checker bedienen, nicht aber unser Ziel zu einem verteilten System zu passen.

Im Allgemeinen können wir nicht davon ausgehen, dass es im Netzwerk bereits eine zentrale Komponente gibt, auf welcher ein sequentieller Checker somit laufen kann. Für einen sequentiellen Checker sind wir folglich auf eine zusätzliche Komponente oder zusätzliche Kanäle angewiesen. Durch die Anforderung der Zentralität an einen sequentiellen Checker (siehe Lemma 6.4.1) ist dessen Integration ins

Netzwerk also eine zusätzliche Herausforderung. Die Checker-Komponente passt nicht zu den Gegebenheiten des ursprünglichen Netzwerks.

Ein zentraler sequentieller Checker erhält die gesamte Eingabe, Ausgabe und den zugehörigen Zeugen jeweils als eine Nachricht jeder Komponente mit ihren entsprechenden Anteilen. Damit spiegelt ein sequentieller Checker nicht den Kerngedanken der Lokalität wider. Die Kommunikation verläuft global und der Checker hat auf alle Informationen Zugriff. Darüber hinaus werden dadurch auch unverhältnismäßig viele Nachrichten an den Checker gesendet.

Die Eingabe ist außerdem mindestens schon das Netzwerk selbst. Dadurch ergeben sich ganz andere Bedingungen an die Checker-Komponente als an eine Komponente des Netzwerks, was die Berechnungsleistung betrifft. Der Kerngedanke der Gleichheit wird entsprechend nicht widergespiegelt.

Weiterhin wird die Verifikationsarbeit nicht auf das Netzwerk verteilt, denn der Checker verifiziert alleine. Das widerspricht dem Kerngedanken der Verteiltheit. Darüber hinaus ist ein zentraler Checker somit auch ein „Bottleneck“ und ein „Single-point-of-failure“ in einem verteilten System.

Wenn es eine Komponente außerhalb eines Netzwerks gibt, die all das leisten kann, stellt sich die Frage, warum diese Komponente nicht auch gleich die gesamte Berechnung des Eingabe-Ausgabe-Paars sequentiell ausführt.

Ein sequentieller Checker müsste nicht zentral sein, wenn es einen Mechanismus gibt durch den die Komponenten ihre Teileingaben, Teilausgaben und Teilzeugen vertrauenswürdig weiterleiten können. In diesem Fall wäre ein sequentieller Checker einfacher ins Netzwerk integrierbar, da er nicht mit allen Komponenten benachbart sein muss. Davon abgesehen sprächen in diesem Fall alle weiteren genannten Gründe weiterhin gegen einen sequentiellen Checker.

6.4.3 Ziel: Verteilte Checker

Wir beschäftigen uns im folgenden nicht mit einer sequentiellen Verifikation zur Laufzeit, sondern mit einer verteilten Verifikation. Unser Ziel sind *verteilte* Checker. Dafür benötigen wir neben einem verteilten Zeugen auch solch ein Zeugenprädikat, das sich durch einen verteilten Checker-Algorithmus entscheiden lässt.

All die beschrieben Nachteile eines sequentiellen Checkers sind Gründe für unser Ziel eines verteilten Checkers. Ein weiterer Grund für einen verteilten Checker ist das in 5.4.3 formulierte Ziel, mit unseren Verifikationsansatz sowohl einem zentralen Nutzer als auch einem ver-

teilten Nutzer eines zertifizierenden verteilten Algorithmus gerecht zu werden. Für einen Nutzer, für den das Eingabe-Ausgabe-Paar ohnehin zentral gesammelt wird, mag der Ansatz eines sequentiellen Checkers noch eine Alternative sein. Zu einem verteilten Nutzer passt ein sequentieller Checker jedoch nicht. Das zentralisierte Einsammeln des Eingabe-Ausgabe-Paares ist dann unnötiger zusätzlicher Aufwand.

Die Abbildung 6.5 zeigt die Architektur für verteilte Checker, wie wir sie anstreben. Wir sehen zum einen ein Netzwerk und zum anderen die Ausführung eines zertifizierenden verteilten Algorithmus aus der Sicht einer Komponente eines Netzwerks. Die Abbildung 6.5 ist eine Erweiterung der Abbildung 5.1. Wir greifen mit der Abbildung 6.5 etwas vor. Hierbei ist unsere Hoffnung, das damit die folgenden Konzepte leichter zu verstehen sind.

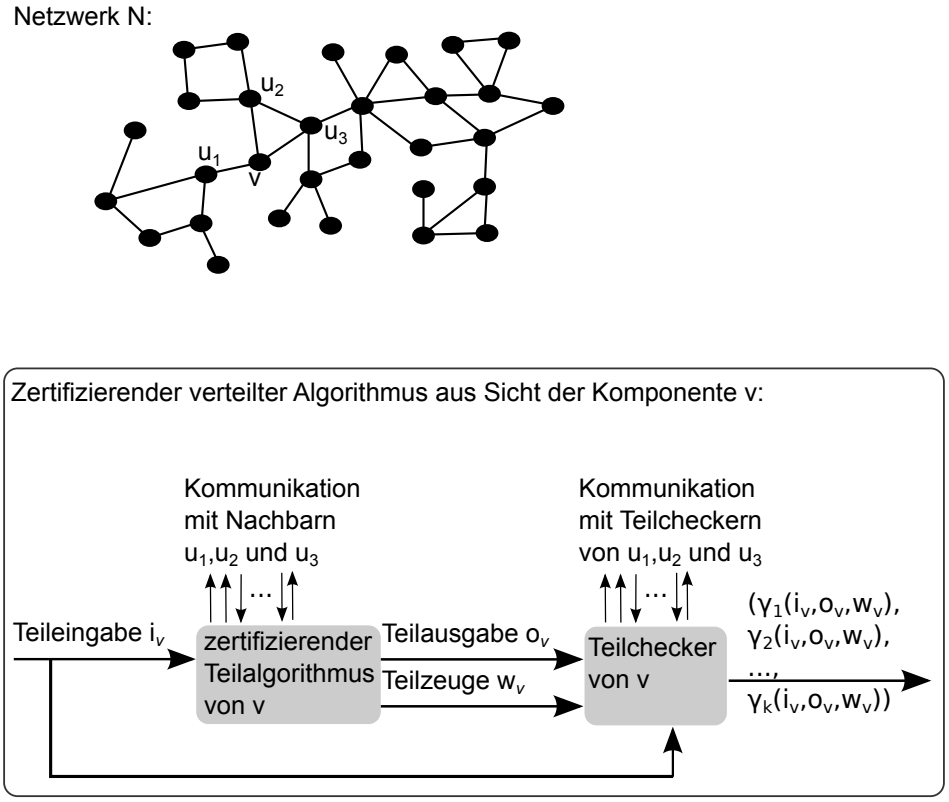


Abbildung 6.5: Im oberen Teil der Abbildung ist ein Netzwerk N mit der Komponente v und ihren Nachbarn u_1, u_2 und u_3 zu sehen. Im unteren Teil des Bilds ist die Ausführung eines zertifizierenden verteilten Algorithmus aus der Sicht der Komponente v dargestellt. Die Kommunikation der Komponente v mit ihren Nachbarn während der Ausführung des Teilalgorithmus ist durch ein- und ausgehende Pfeile dargestellt. Genauso verhält es sich mit der Kommunikation des Teilcheckers. Der Teilchecker entscheidet lokale Prädikate $(\gamma_1, \gamma_2, ..., \gamma_k)$ für seine Komponente.

6.5 VERTEILBARE PRÄDIKATE

Wir führen in diesem Abschnitt verteilbare Prädikate ein. Ein verteilbares Prädikat ist ein globales Prädikat in einem Netzwerk, das entschieden werden kann, indem lokale Prädikate für die Komponenten entschieden werden. Diese verteilte Entscheidung passt zu einem Netzwerk. Wir haben verteilbare Prädikate in [Völ17] vorgestellt und somit einige Ergebnisse dieses Abschnitts veröffentlicht.

In den Definitionen dieses Abschnitts gehen wir von *konsistenten* und *wohlgeformten* Zeugen aus. Wir behandeln die beiden Konzepte jedoch erst in Kapitel 7 zu verteilten Zeugen. Der Grund dafür ist, dass die Betrachtungen zum verteilten Zeugen vor der Einführung verteilbarer Zeugenprädikate uns recht technisch erscheinen. Wir geben in Abschnitt 6.5.1 deshalb eine Intuition.

In Abschnitt 6.5.2 führen wir universell-verteilbare Prädikate ein. Damit ein universell-verteilbares Prädikat in einem Netzwerk erfüllt ist, muss ein lokales Prädikat für *alle* Komponenten des Netzwerks gelten.

In Abschnitt 6.5.3 führen wir existenziell-verteilbare Prädikate ein. Damit ein existenziell-verteilbares Prädikat in einem Netzwerk erfüllt ist, muss ein lokales Prädikat für *mindestens eine* Komponente des Netzwerks gelten.

In Abschnitt 6.5.4 führen wir dann schließlich ein verteilbares Prädikat ein – eine Kombination universell-verteilbarer und existenziell-verteilbarer Prädikate.

In Abschnitt 6.5.5 diskutieren wir abschließend Vorteile und Nachteile verteilbarer Zeugenprädikate.

6.5.1 Intuition: konsistente und wohlgeformte Zeugen

Die Idee eines verteilbaren Prädikats kann unserer Auffassung nach zunächst auch gut ohne die Konzepte der Konsistenz und der Wohlgeformtheit eines Zeugen verstanden werden. Dennoch geben wir eine Intuition.

Wir haben in unserem einführenden Beispiel bereits einen konsistenten (und wohlgeformten) Zeugen kennengelernt (siehe Abschnitt 6.1). Bei einem verteilten Zeugen ist es häufig so, dass sich einige Teilzeugen Informationen teilen. Wir haben in unserem einführenden Beispiel gesehen, dass der aufgezeigte Zeuge nur dann ein schlüssiges Korrektheitsargument für ein Eingabe-Ausgabe-Paar ist, wenn sich seine Teilzeugen in den geteilten Informationen nicht widersprechen. Ein Zeuge, dessen Teilzeugen widerspruchsfrei sind, ist konsistent. Das ist der Grund, warum wir uns auf konsistente Zeugen beziehen.

Das Konzept der Wohlgeformtheit eines Zeugen wiederum ist dafür nötig, dass wir Betrachtung der Konsistenz auf Zeugen beschränken können und nicht auch eine Widerspruchsfreiheit für die verteilte Eingabe und Ausgabe fordern müssen.

6.5.2 Universell-verteilbare Prädikate

Ein globales Prädikat ist über einer Eingabe, einer Ausgabe und/oder einem Zeugen definiert. Das heißt, wir machen keinen Gebrauch von Teileingaben, Teilausgaben oder Teilzeugen der Komponenten. Mit lokalen Prädikaten können wir jedoch auch Eigenschaften für die einzelnen Komponenten formulieren und diese in Relation setzen zu einer Eigenschaft des Netzwerks.

Wir bezeichnen ein globales Prädikat, welches gilt, wenn für *alle* Komponenten eine Eigenschaft lokal erfüllt ist, als ein *universell-verteilbares* Prädikat:

Definition 18 (universell-verteilbares Prädikat)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$ *seien* I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Sei $P \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein globales Prädikat in* N. *Seien die Prädikate* $p_v \subseteq [I_v] \times [O_v] \times [W_v]$ *für alle* $v \in V$ *lokal in* N.

P *ist* universell-verteilbar *in* N *mit* p_v, *falls für alle* $i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!]$ *mit* w *wohlgeformt und konsistent gilt:*

$$(\forall v \in V : p_v(i_v, o_v, w_v)) \longrightarrow P(i, o, w) \tag{6.2}$$

Wir bezeichnen die Eigenschaft (6.2) als *Verteilungseigenschaft* eines globalen Prädikats.

Beispiel 9 (Netzwerk ist bipartit)**.** *Aus dem Beispiel des Bipartitheitstests kennen wir bereits ein universell-verteilbares Prädikat. (siehe Abschnitt 6.1). Eine Bipartition in einem Netzwerk ist ein Zeuge dafür, dass das Netzwerk bipartit ist. Jede Komponente hat als Teilzeugen eine Bipartition ihrer Nachbarschaft. Das zugehörige Zeugenprädikat ist erfüllt, wenn der potenzielle Zeuge tatsächlich eine Bipartition ist.*

Dieses Zeugenprädikat ist universell-verteilbar mit einem lokalen Prädikat, das für eine Komponente erfüllt ist, wenn der potenzielle Teilzeuge tatsächlich eine Bipartition der Nachbarschaft ist. Wenn das lokale Prädikat für alle *Komponenten erfüllt ist, dann gibt es in jeder Nachbarschaft eine Bipartition. Die überlappenden Bipartitionen der Nachbarschaften bilden gemeinsam eine Bipartition des Netzwerks. Damit ist das Zeugenprädikat entsprechend erfüllt. Das Zeugenprädikat ist also universell-verteilbar.*

In der Definition eines universell-verteilbaren Prädikats beziehen wir uns auf ein globales Prädikat mit der Signatur eines Zeugenprädikats.

Für ein globales Prädikat anderer Signatur sei universell-verteilbar analog definiert (siehe die Definition 11 für ein globales Prädikat). Für die Definitionen für existenziell-verteilbare Prädikate und für verteilbare Prädikate werden wir in gleicher Weise verfahren.

In dem Beispiel 9 sind die lokalen Prädikate für alle Komponenten generisch definiert: für jede Komponente ist ihr lokales Prädikat erfüllt, wenn eine Färbung ihrer Nachbarschaft eine Bipartition ist. Da alle Komponenten den gleichen Teilalgorithmus ausführen und sich nur durch Position oder eine Sonderrolle unterscheiden, ist dies der Normalfall.

Wir erinnern uns daran, dass es für einen Nutzer eines zertifizierenden Algorithmus ausreicht, wenn das Zeugenprädikat mit seinem konkreten Eingabe-Ausgabe-Paar und Zeugen erfüllt ist. Hingegen ist es für den Nutzer irrelevant, ob es zu jedem korrekten Eingabe-Ausgabe-Paar einen entsprechenden Zeugen gibt. Genauso wie die Zeugeneigenschaft deswegen eine Implikation ist, ist auch die Verteilungseigenschaft eine Implikation. Es gilt deshalb:

Lemma 6.5.1. *Jedes globale Prädikat ist universell-verteilbar in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen mit einem lokalen Prädikat für jede Komponente des Netzwerks, das nie erfüllt ist.*

Beweis. Folgt aus der Definition 19 direkt aus der Verteilungseigenschaft (6.2). □

Die Entwicklerin eines zertifizierenden verteilten Algorithmus hingegen muss für ein universell-verteilbares Zeugenprädikat sicherstellen, dass es immer durch entsprechende lokale Prädikat entschieden werden kann. Wir definieren deswegen *vollständig* universell-verteilbare Prädikate.

6.5.2.1 *Vollständig universell-verteilbare Prädikate*

Wir bezeichnen ein globales Prädikat, welches genau dann gilt, wenn für *alle* Komponenten eine Eigenschaft lokal gilt, als ein vollständig universell-verteilbares Prädikat.

Definition 19 (vollständig universell-verteilbares Prädikat)**.** *Seien* N, I, O, W, $\mathtt{Val}_I$, $\mathtt{Val}_O$, $\mathtt{Val}_W$, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$: I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Seien P *und* p_v *für alle* $v \in V$ *wie in Definition 19 gegeben. Sei* P *universell-verteilbar in* N *mit* p_v *für alle* $v \in V$.

P *ist* vollständig *universell-verteilbar, falls die Eigenschaft* (6.2) *aus Definition 19 als Biimplikation gilt.*

Beispiel 10 (Netzwerk ist bipartit)**.** *Das Prädikat des vorherigen Beispiels 9 ist vollständig universell-verteilbar. Wenn es eine Bipartition in einem Netzwerk gibt, dann gibt es auch Bipartitionen der Nachbarschaften.*

Lemma 6.5.2. *Nicht jedes globale Prädikat ist vollständig universell-verteilbar in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen.*

Beweis. Sei $N = (V, E)$ ein Netzwerk. Sei $I = \{id_v | v \in V\} \cup \{nbrs_v | v \in V\}$ und für alle $v \in V$ sei $I_v = \{id_v, nbrs_v\}$ die Variablenmenge der Eingaben bzw. Teileingaben der Komponenten. Sei $Val_I = V \cup P(V)$ die Wertemenge der Eingaben und Teileingaben. Sei $P \subseteq [\![I]\!]$ ein globales Prädikat, das erfüllt ist, falls es eine Komponente $v \in V$ gibt, sodass $id_v = |nbrs_v|$.

Für das globale Prädikat P in N gibt es keine lokalen Prädikate für die Komponenten mit denen P vollständig universell-verteilbar ist. Nehmen wir an P gilt in N mit einer Eingabe, dann muss für jede Komponente ihr lokales Prädikat erfüllt sein. Das heißt auch für alle Komponenten $v \in V$ für deren Teileingabe gilt: $id_v \neq |nbrs_v|$.

Nehmen wir nun an P gilt für eine Eingabe in N nicht, so gilt für alle Komponenten $v \in V$: $id_v \neq |nbrs_v|$. Folglich sind nun für alle Komponenten die lokalen Prädikate auch erfüllt. Das ist ein Widerspruch dazu, dass P vollständig universell-verteilbar ist. □

6.5.3 Existenziell-verteilbare Prädikate

Ein globales Prädikat, das gilt, wenn für *mindestens eine* Komponente eine Eigenschaften gilt, ist ein *existenziell-verteilbares* Prädikat.

Definition 20 (existenziell-verteilbares Prädikat)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$ *seien* I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Sei $P \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein globales Prädikat in* N. *Für alle* $v \in V$ *seien die Prädikate* $p_v \subseteq [I_v] \times [O_v] \times [W_v]$ *lokal in* N.

P *ist* existenziell-verteilbar *in* N *mit* p_v *für alle* $v \in V$, *falls für alle* $i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!]$ *mit* w *wohlgeformt und konsistent gilt:*

$$(\exists v \in V : p_v(i_v, o_v, w_v)) \longrightarrow P(i, o, w) \tag{6.3}$$

Die Eigenschaft (6.3) ist die Verteilungseigenschaft eines existenziell-verteilbaren Prädikats.

Beispiel 11 (Keine Bipartition eines Netzwerks)**.** *Nehmen wir an, die Eingabe eines verteilten Algorithmus ist das Netzwerk auf dem er läuft und die Ausgabe eine Färbung der Komponenten mit zwei Farben.*

Wir nehmen ein globales Prädikat über der Eingabe und Ausgabe an, das erfüllt ist, wenn die Färbung keine Bipartition ist. Ein Paar benachbarter Komponenten mit gleicher Farbe bezeugt, dass die Färbung keine Bipartition des Netzwerks ist.

Sei die Teilausgabe einer Komponente die Färbung ihrer Nachbarschaft. Dann ist das globale Prädikat existenziell-verteilbar mit lokalen Prädikaten je Komponente, die für ihre Komponente erfüllt sind, falls einer ihrer Nachbarn die gleiche Farbe hat.

Zwei Nachbarn gleicher Farbe bezeugen also, dass eine Färbung keine Bipartition eines Netzwerks ist. Wir weisen darauf hin, dass diese Eigenschaft nicht ausreicht um zu bezeugen, dass ein Netzwerk nicht bipartit ist. Schließlich könnte es weiterhin eine Bipartition des Netzwerks geben.

Lemma 6.5.3. *Jedes globale Prädikat ist existenziell-verteilbar in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen mit einem lokalen Prädikat je Komponente, das nie erfüllt ist.*

Beweis. Folgt aus der Definition 20 direkt aus der Verteilungseigenschaft (6.3). □

6.5.3.1 *Vollständig existenziell-verteilbare Prädikate*

Wie wir bereits universell-verteilbare Prädikate definiert haben, definieren wir nun analog vollständig existenziell-verteilbare Prädikate.

Definition 21 (vollständig existenziell-verteilbares Prädikat)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$ *seien* I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Seien* P *und* p_v *für alle* $v \in V$ *wie in Definition 20 gegeben. Sei* P *existenziell-verteilbar in* N *mit* p_v *für alle* $v \in V$.

P *ist* vollständig *existenziell-verteilbar, falls die Eigenschaft* (6.3) *aus der Definition 20 als Biimplikation gilt.*

Beispiel 12 (Keine Bipartition eines Netzwerks)**.** *Das globale Prädikat des vorherigen Beispiels 11 ist vollständig existenziell-verteilbar. Es gibt immer genau dann zwei Nachbarn gleicher Farbe, wenn die Färbung keine Bipartition ist.*

Wohlgemerkt ist das Prädikat vollständig existenziell-verteilbar, weil wir von nur zwei Farben ausgehen. Gäbe es mehr als zwei Farben, könnte eine Färbung auch aufgrund einer weiteren Farbe keine Bipartition sein.

In Lemma 6.5.2 haben wir gezeigt, dass nicht jedes globale Prädikat vollständig universell-verteilbar ist und genauso ist auch nicht jedes globale Prädikat vollständig existenziell-verteilbar in einem Netzwerk.

Lemma 6.5.4. *Nicht jedes globale Prädikat ist in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen vollständig existenziell-verteilbar.*

Beweis. Sei $N = (V, E)$ ein Netzwerk. Sei $I = \{id_v | v \in V\} \cup \{nbrs_v | v \in V\}$ und für alle $v \in V$ sei $I_v = \{id_v, nbrs_v\}$ die Variablenmenge der Eingaben bzw. Teileingaben der Komponenten. Sei $Val_I = V \cup P(V)$ die Wertemenge der Eingaben und Teileingaben. Sei $P \subseteq [\![I]\!]$ ein globales Prädikat, das erfüllt ist, falls für alle Komponenten $v \in V$ gilt: $id_v = |nbrs_v|$.

Für das globale Prädikat P in N gibt es keine lokalen Prädikate der Komponenten, sodass P vollständig existenziell-verteilbar ist. Nehmen wir an P gilt für nicht für eine Eingabe in N. Dann kann es dennoch eine Komponente $v \in V$, geben für die gilt: $id_v = |nbrs_v|$. Für keine Komponente darf ihr lokales Prädikat erfüllt sein und somit kann es auch nicht für die Teileingabe einer Komponente erfüllt sein, für die die Eigenschaft gilt.

Nehmen wir nun an P gilt für eine Eingabe in N, so gilt für alle Komponenten $v \in V$: $id_v = |nbrs_v|$. Folglich ist für keine der Komponente das lokale Prädikat erfüllt. Das ist ein Widerspruch dazu, dass P vollständig existenziell-verteilbar ist. □

6.5.4 Definition verteilbarer Prädikate

Wir verallgemeinern die universelle und existenzielle Verteilbarkeit eines globalen Prädikats in einem Netzwerk und definieren ein *verteilbares* Prädikat.

Definition 22 (verteilbares Prädikat)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$ *seien* I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Sei $P \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein globales Prädikat in* N. *Seien die Prädikate* $p_v \subseteq [I_v] \times [O_v] \times [W_v]$ *für jedes* $v \in V$ *lokal in* N.

P *ist* verteilbar *in* N, *falls eine der folgenden Bedingungen gilt:*

1. P *ist universell-verteilbar.*
2. P *ist existenziell-verteilbar.*
3. *Es gibt ein verteilbares globales Prädikat* P_1, *sodass für alle* $i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!]$ *mit* w *wohlgeformt und konsistent gilt:*

$$\neg P_1(i, o, w) \longrightarrow P(i, o, w)$$

4. *Es gibt verteilbare globale Prädikat* P_1, P_2, *sodass für alle* $i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!]$ *mit* w *wohlgeformt und konsistent gilt:*

$$P_1(i, o, w) \wedge P_2(i, o, w) \longrightarrow P(i, o, w)$$

5. *Es gibt verteilbare globale Prädikat* P_1, P_2, *sodass* $i \in [\![I]\!], o \in [\![O]\!], w \in [\![W]\!]$ *mit* w *wohlgeformt und konsistent gilt:*

$$P_1(i, o, w) \vee P_2(i, o, w) \longrightarrow P(i, o, w)$$

In den Bedingungen 3., 4., und 5. führen wir zusätzliche globale Prädikate (P_1 und P_2) ein, um ein globales Prädikat verteilbar zu gestalten. Wir bezeichnen diese globalen Prädikate als die *Verteilungsprädikate* des verteilbaren Prädikats.

Beispiel 13. *Die globalen Prädikate, die wir als Beispiel für ein universell-verteilbares und für ein existenziell-verteilbares Prädikat aufgeführt haben, sind jeweils auch verteilbar, da sie der Bedingung 1. beziehungsweise der Bedingung 2. genügen. Siehe Beispiel 9 beziehungsweise Beispiel 11.*

In den Fallstudien dieser Arbeit betrachten wir auch verteilbare Zeugenprädikate, die nicht mit universeller oder existenzieller Verteilbarkeit zusammenfallen.

Lemma 6.5.5. *Jedes globale Prädikat ist in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen verteilbar.*

Beweis. Folgt aus dem Lemma 6.5.1 und dem Lemma 6.5.3. □

6.5.4.1 *Vollständig verteilbare Prädikat*

Wir definieren ein vollständig verteilbares Prädikat.

Definition 23 (vollständig verteilbares Prädikat)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$: I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $P \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein verteilbares globales Prädikat in* N *mit den Verteilungsprädikaten* $P_1, P_2, ..., P_j$.

Das globale Prädikat P *ist* vollständig *verteilbar in* N, *falls die Prädikate* $P_1, P_2, ..., P_j$ *jeweils vollständig universell-verteilbar oder vollständig existenziell-verteilbar sind und die Bedingungen 3.-5. der Definition 22 für* P *als Biimplikation gelten.*

Beispiel 14. *Das universell-verteilbare Prädikat aus dem Beispiel 10 und das existenziell-verteilbare Prädikat aus dem Beispiel 12 sind jeweils ein Beispiel für ein vollständig verteilbares Prädikat.*

Lemma 6.5.6. *Nicht jedes globale Prädikat ist vollständig verteilbar in einem beliebigen Netzwerk für beliebige Eingaben, Ausgaben und potenziellen Zeugen.*

Beweis. Sei $N = (V, E)$ ein Netzwerk. Sei $I = \{id_v | v \in V\} \cup \{nbrs_v | v \in V\}$ und für alle $v \in V$ sei $I_v = \{id_v, nbrs_v\}$ die Variablenmenge

der Eingaben beziehungsweise der Teileingaben je Komponente. Sei $\mathrm{Val}_I = V \cup P(V)$ die Wertemenge der Eingaben sowie Teileingaben. Sei $P \subseteq [\![I]\!]$ ein globales Prädikat, das erfüllt ist, falls es Komponenten $u, v \in V$ gibt, sodass $\mathrm{id}_u = |\mathrm{nbrs}_v|$.

Als Teileingabe hat eine Komponente also ihre eigene ID und ihre Nachbarn. Entsprechend hat keine Komponente Informationen darüber, welche Nachbarn eine andere Komponente hat. Es gibt deswegen kein lokales Prädikat über der Teileingabe einer Komponente, das in Abhängigkeit der obigen Eigenschaft erfüllt ist. □

Aus dem Beweis wird deutlich, dass es ein vollständig verteilbares Prädikat für die genannte Eigenschaft geben kann, wenn wir die Teileingaben anders wählen. Die Eingabe und Ausgabe eines verteilten Algorithmus ist durch den Algorithmus festgelegt. Die Entwicklerin einer zertifizierenden Variante des Algorithmus bestimmt jedoch sowohl das Zeugenprädikat als auch den Zeugen. Die Informationen, die in einem Teilzeugen einer Komponente angesammelt werden, beeinflussen dann welche Eigenschaften durch lokale Prädikate ausgedrückt werden können. Somit hat die Wahl der Zeugen einen Einfluss auf die Verteilbarkeit des Zeugenprädikats.

6.5.5 Vorteile und Nachteile verteilbarer Zeugenprädikate

6.5.5.1 *Vorteile*

Ein Vorteil eines verteilbaren Zeugenprädikats besteht darin, dass die Verifikation durch die lokalen Prädikate (relativ) gleichmäßig auf die Komponenten eines Netzwerks verteilt werden kann. Hierdurch werden die Kerngedanken der Verteiltheit und der Gleichheit beachtet.

Darüber hinaus entscheidet jede Komponente ihre lokalen Prädikate ohne weitere Kommunikation mit den anderen Komponenten des Netzwerks. Hiermit wird der Kerngedanke der Lokalität widergespiegelt.

Ein verteilbares Zeugenprädikat ermöglicht also einen verteilten Checker, der die Kerngedanken der Verteiltheit, Gleichheit und Lokalität widerspiegelt. Inwieweit ein Checker die Kerngedanken widerspiegelt, hängt jedoch auch von der Wahl der Zeugen ab; wir beschäftigen uns im folgenden Kapitel 7 ausführlich mit verteilten Zeugen.

6.5.5.2 *Nachteile*

Die logische Struktur, die ein verteilbares Prädikat haben kann, ist eingeschränkt (siehe Definition 22 eines verteilbaren Prädikats). Zum Beispiel ist eine Verschachtelung von Quantoren nicht möglich.

Diese Einschränkung ist jedoch motiviert durch unser Ziel eines verteilten Checkers. So werden wir sehen, dass mit einem verteilbaren Zeugenprädikat die Kommunikation für die Kombination der ausgewerteten lokalen Prädikate nach immer dem gleichen Muster abläuft unabhängig von dem jeweiligen Zeugenprädikat (siehe Teil iv).

7 VERTEILTE ZEUGEN

Ein Zeuge wird *verteilt* berechnet und liegt *verteilt* in einem Netzwerk vor. Durch die Verteiltheit eines Zeugen ergeben sich einige Unterschiede zwischen Zeugen zertifizierender verteilter Algorithmen und Zeugen zertifizierender sequentieller Algorithmen. Wir beschäftigen uns in diesem Kapitel genauer mit den Zeugen zertifizierender verteilter Algorithmen. Hierbei betrachten wir Eigenschaften potenzieller Zeugen, sprechen dennoch verkürzt von Zeugen, ohne uns jedoch auf ein Eingabe-Ausgabe-Paar zu beziehen.

In Abschnitt 7.1 führen wir zentralisierte, gleichverteilte und beschränkte Zeugen ein und diskutieren an ihnen die Kerngedanken der Verteiltheit, Gleichheit und Lokalität.

Für einen verteilten Zeugen teilen sich einige Teilzeugen häufig Informationen. In Abschnitt 7.2 erläutern wir, warum wir Teilzeugen im Allgemeinen so wählen, dass sie sich Informationen teilen.

Ein Zeuge ist nur dann ein Korrektheitsargument, wenn sich seine Teilzeugen in geteilten Informationen nicht widersprechen. In Abschnitt 7.3 führen wir konsistente Zeugen ein. Einige der Resultate dieses Abschnitts haben wir in [VA18] veröffentlicht.

Für eine bessere Lesbarkeit nehmen wir in diesem Kapitel wieder die Objekte eines festen, aber beliebigen Interfaces eines zertifizierenden verteilten Algorithmus an. Zur Erinnerung siehe Abschnitt 6.2.3 auf Seite 66 des vorigen Kapitels. Dort listen wir die Objekte mit ihrer Bedeutung zum einfachen Nachschlagen in der Abbildung 6.4 auf und verweisen außerdem auf alle entsprechenden Definitionen.

7.1 VERTEILTHEIT, GLEICHHEIT UND LOKALITÄT BEI ZEUGEN

Eine Entwicklerin einer zertifizierenden Variante eines verteilten Algorithmus wählt neben dem Zeugenprädikat auch die Zeugen. Dabei legt sie unter anderem fest, wie ein Zeuge auf die Komponenten verteilt wird. Letztlich entscheidet sie damit darüber, in welchem Grad der Zeuge die Kerngedanken der Verteiltheit, Gleichheit und Lokalität beachtet. Wir diskutieren in diesem Abschnitt welche Eigenschaften für einen verteilten Zeugen wünschenswert sind.

In Abschnitt 7.1.1 stellen wir *zentralisierte* Zeugen vor. Ein Zeuge, der nicht zentralisiert ist, spiegelt den Kerngedanken der Verteiltheit wider. In Abschnitt 7.1.2 beschäftigen wir uns mit *gleichverteilten* Zeugen, bei denen jeder Teilzeuge den gleichen Beitrag zum Korrektheitsargument leistet. Ein gleichverteilter Zeuge spiegelt also den Kerngedanken der Gleichheit wider. In Abschnitt 7.1.3 führen wir *beschränkte* Zeugen ein, die den Kerngedanken der Lokalität widerspiegeln. In Abschnitt 7.1.4 erläutern wir welche Zeugen wir anstreben.

7.1.1 Zentralisierte Zeugen

Ein Zeuge ist *zentralisiert*, wenn es eine Komponente gibt, deren Teilzeuge *alle* Informationen des Zeugen besitzt. Wir definieren zentralisierte Zeugen wie folgt:

Definition 24 (zentralisierter Zeuge)**.** *Seien* N, W, $\mathtt{Val}_W$, $[\![W]\!]$, *sowie für alle* $v \in V$ *seien* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

w *ist* zentralisiert, *falls es eine Komponente* $v \in V$ *gibt, sodass* $W_v = W$.

Beispiel 15. *Erinnern wir uns an das einführende Beispiel des zertifizierenden Bipartitheitstests (siehe Abschnitt 6.1); jede Komponente hat als Teilzeugen eine Bipartition ihrer Nachbarschaft.*

Nehmen wir nun als Variante einen Zeugen an, bei dem eine Komponente als Teilzeugen eine Bipartition des gesamten Netzwerks hat. Dieser Zeuge ist dann zentralisiert.

Bei einem zentralisierten Zeugen liegen alle Informationen des Zeugen bei mindestens einer Komponente vor. Ein zentralisierter Zeuge hat deswegen einen geringen Grad an Verteiltheit.

7.1.1.1 *Vorteile zentralisierter Zeugen*

Ein Vorteil eines zentralisierten Zeugen ist, dass er dem Zeugen eines zertifizierenden sequentiellen Algorithmus nahe kommt. Bei einem Zeugen eines zertifizierenden sequentiellen Algorithmus liegen auch alle Informationen des Zeugen an einer Stelle vor, nämlich dem Zeugen selbst. Bei einem zentralisierten Zeugen gibt es einen Teilzeugen, der so aufgebaut ist.

Ein weiterer Vorteil ist, dass ein zentralisierter Zeuge meist verständlicher ist; das Korrektheitsargument, das er repräsentiert, liegt vollständig an einer Stelle vor. Für einen Nutzer ist das Korrektheitsargument so im Allgemeinen verständlicher.

7.1.1.2 *Nachteile zentralisierter Zeugen*

Ein zentralisierter Zeuge widerspricht dem Kerngedanken der Verteiltheit und hat deshalb einige Nachteile. Auch ein zentralisierter Zeuge wird durch einen zertifizierenden verteilten Algorithmus berechnet. Es liegt auf der Hand, dass die Zentralisierung eines Zeugen im Widerspruch zu einer verteilten Berechnung eines Zeugen steht.

Ein weiterer Nachteil ist, dass im Allgemeinen mindestens ein Teilzeuge unverhältnismäßig groß ist im Vergleich zu den anderen Teilzeugen, aber auch im Vergleich zu Teileingaben und Teilausgaben. Das steht im Widerspruch zum Kerngedanken der Gleichheit.

Ein zentralisierter Zeuge beeinflusst auch, wie ein Checker aussehen kann. Für einen zentralisierten Zeugen liegt es nahe, dass die Entscheidung des Zeugenprädikats von einem zentralen sequentiellen Checker übernommen wird. In diesem Fall ergeben sich jedoch alle Nachteile, die wir bereits für einen solchen Checker in Abschnitt 6.4 diskutiert haben.

Eine andere Variante ist die von uns angestrebte verteilte Checker-Architektur (siehe Abbildung 6.5 auf Seite 75). In diesem Fall würde ein Teilchecker jedoch die gesamte Verifikationsarbeit leisten und auch das widerspricht dem Kerngedanken der Gleichheit.

7.1.2 Gleichverteilte Zeugen

Ein *gleichverteilter* Zeuge zeichnet sich dadurch aus, dass seine Teilzeugen, bis auf Unterschiede durch die Position einer Komponente, gleich aufgebaut sind.

Wir führen dafür eine *topologische Substitution* von Variablen einer Komponente in Abhängigkeit der Topologie eines Netzwerks ein: Alle Variablen einer Komponente, die sich auf ihre Position im Netzwerk beziehen, substituieren wir mit den entsprechenden Variablen einer anderen Komponente. Wichtig ist dabei, dass sich die Anzahl der Variablen ändern können.

Notation (topologische Substitution)**.** *Wir kennzeichnen die Substitution der positionsabhängigen Variablen einer Komponente v_2 durch die einer Komponente v_1 mit $v_2 \hookleftarrow v_1$.*

Beispiel 16. *Sei für jede Komponente v eines Netzwerks die Variablenmenge $I_v = \{col_u | u \text{ ist Nachbar von } v\} \cup \{id_v\} \cup \{\text{leader}\}$ definiert. Seien die Variablenmengen für die Komponenten a, b wie folgt gegeben: $I_a = \{col_b, col_c, col_d\} \cup \{id_a\} \cup \{\text{leader}\}$ und $I_b = \{col_a, col_d\} \cup \{id_b\} \cup \{\text{leader}\}$. Dann sieht die Variablenmenge $I_{a \hookleftarrow b}$ wie folgt aus $I_{a \hookleftarrow b} = \{col_a, col_d\} \cup \{id_b\} \cup \{\text{leader}\} = I_b$.*

Wir definieren gleichverteilte Zeugen wie folgt:

Definition 25 (gleichverteilter Zeuge)**.** *Seien* N, W, $\mathtt{Val}_W$, $[\![W]\!]$, *sowie für alle* $v \in V$ *seien* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

w *ist* gleichverteilt, *falls für alle* $u, v \in V$ *mit* $u \neq v$ *gilt:* $W_u = W_{v \leftrightarrow u}$.

Beispiel 17. *In dem Beispiel des zertifizierenden Bipartitheitstests (siehe Abschnitt 6.1) enthält der Teilzeuge einer jeden Komponente eine Bipartition der Nachbarschaft. Die Teilzeugen sind folglich bis auf Unterschiede durch die Position im Netzwerk gleich aufgebaut. Der Zeuge, den wir in Abschnitt 6.1 vorgestellt haben, ist somit gleichverteilt.*

7.1.2.1 *Vorteile gleichverteilter Zeugen*

Ein gleichverteilter Zeuge ist so aufgebaut, dass jeder Teilzeuge einen gleichen Beitrag zum Korrektheitsargument leistet. Für jede Komponente sieht die Berechnung ihres Teilzeugen folglich auch gleich aus. Ein gleichverteilter Zeuge passt außerdem zu der von uns angestrebten verteilten Checker-Architektur (siehe Abbildung 6.5 auf Seite 75). Jeder Teilchecker ist dabei entsprechend auch gleich aufgebaut. Somit spiegelt ein gleichverteilter Zeuge den Kerngedanken der Gleichheit wider.

7.1.2.2 *Nachteile gleichverteilter Zeugen*

Ein gleichverteilter Zeuge kann auch zentralisiert sein. Nämlich dann, wenn der Teilzeuge einer jeden Komponenten jeweils alle Informationen besitzt. Die Informationen sind dann bei jeder Komponente zentralisiert und alle Informationen werden global geteilt. Das widerspricht dem Kerngedanken der Verteiltheit und der Lokalität.

7.1.3 Beschränkte Zeugen

Ein *beschränkter* Zeuge zeichnet sich dadurch aus, dass Information zwischen Nachbarn geteilt werden. Dabei lassen wir zu, das eine Komponente Informationen mit Nachbarn und auch noch Nachbars Nachbarn teilt. Der Grund ist, dass wir erlauben wollen, dass eine Komponente eine Information mit all ihren Nachbarn teilt. Informationen zwischen Teilzeugen werden deshalb höchstens in 2-Nachbarschaften geteilt. So funktionieren bereits viele Zeugen. Wir diskutieren folgend eine Erweiterung auf k-Nachbarschaften. Wir definieren beschränkte Zeugen wie folgt:

Definition 26 (beschränkter Zeuge)**.** *Seien* N, W, $\mathtt{Val}_W$, $[\![W]\!]$, *sowie für alle* $v \in V$ *seien* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

w ist beschränkt, *falls für alle* $u, v \in V$ *mit* $u \neq v$ *gilt: wenn* u *und* v *keine Nachbarn sind, dann* $W_u \cap W_v = \emptyset$ *oder für alle* $a \in W_u \cap W_v$ *gibt es* $x \in V$, *sodass* x *Nachbar von* u *und von* v *ist und* $a \in W_x$.

Beispiel 18. *Der Zeuge, den wir in dem einführenden Beispiel des zertifizierenden Bipartitheitstests vorgestellt haben (siehe Abschnitt 6.1), ist beschränkt. Denn Teilzeugen von Komponenten, die weder benachbart sind, noch gemeinsame Nachbarn haben, teilen sich keine Informationen. Für Komponenten mit einem gemeinsamen Nachbarn gilt, dass beide jeweils eine Information zur Farbe des gemeinsamen Nachbarn in ihren Teilzeugen enthalten. Der gemeinsame Nachbar enthält die entsprechende Information in seinem Teilzeugen. Informationen werden nicht über die 2-Nachbarschaft hinaus geteilt. Der Zeuge ist also beschränkt.*

7.1.3.1 *Vorteile beschränkter Zeugen*

Informationen werden nicht global, sondern nur lokal in Nachbarschaften geteilt. Für eine verteilte Berechnung eines Zeugen liegt außerdem nahe, dass eine Komponente für die Berechnung ihres Teilzeugen entsprechend auch nicht über ihre 1-Nachbarschaft hinaus kommunizieren muss. Damit spiegelt ein beschränkter Zeuge den Kerngedanken der Lokalität wider.

7.1.3.2 *Nachteile beschränkter Zeugen*

Beschränkte Zeugen sind häufig nicht möglich. So stellen wir zum Beispiel in Abschnitt 9.3 eine zertifizierende Konstruktion eines gewurzelten Spannbaums über den Komponenten eines Netzwerk vor. Für die Zertifizierung muss die ID der Wurzel als Information in jedem Teilzeugen enthalten sein, um auszuschließen, dass es sich um einen Wald handelt. Die ID der Wurzel ist dann folglich eine global geteilte Information. Damit ist dieser Zeuge nicht beschränkt.

Wir begegnen Spannbäumen bei verteilten Algorithmen häufig. Zum einen ermöglicht ein Spannbaum lokale Berechnungen zu einer globalen Berechnung zusammenzuführen [Ray13, Kapitel 3] und zum anderen ermöglicht er eine Broadcast-Kommunikation [Ray13, Kapitel 1]. Beides grundlegende Aspekte, wenn verteilte Komponenten gemeinsam ein Problem lösen.

Darüber hinaus kann ein beschränkter Zeuge auch zentralisiert sein und widerspricht in diesem Fall dem Kerngedanken der Verteiltheit. Teilzeugen können außerdem sehr ungleich auf Komponenten verteilt sein, wodurch der Kerngedanke der Gleichheit verletzt wird.

7.1.3.3 *Erweiterung auf* k-*Nachbarschaft*

Die Definition eines beschränkten Zeugen ist durch die Beschränkung auf eine 2-Nachbarschaft recht restriktiv. Eine Möglichkeit, einen beschränkten Zeugen etwas allgemeiner zu fassen, ist eine Beschränkung auf eine k-Nachbarschaft statt auf eine 2-Nachbarschaft. Die Konstante k ist dann von dem zu lösenden Problem abhängig. Diese Erweiterung ist dann zwar weniger strikt, ändert jedoch nichts generell an den aufgezeigten Vorteilen und Nachteilen.

7.1.4 Fazit: Verteiltheit, Gleichheit, Lokalität bei Zeugen

Wir streben solche Zeugen an, die die Kerngedanken der Verteiltheit, Gleichheit und Lokalität widerspiegeln. Ein nicht zentralisierter, gleichverteilter und beschränkter Zeuge spiegelt die Kerngedanken wider und ist folglich ein wünschenswerter Zeuge. Die Berechnung eines Zeugen ist Aufgabe eines zertifizierenden verteilten Algorithmus. Ein nicht zentralisierter, gleichverteilter und beschränkter Zeuge kann so berechnet werden, dass die Kerngedanken beachtet werden. Für den Nutzer ist der zertifizierende verteilte Algorithmus jedoch eine Black-Box und somit ist es aus der Nutzerperspektive auch irrelevant, wie ein Zeuge berechnet wird.

Die Wahl eines Zeugen ist eine Design-Entscheidung der Entwicklerin einer zertifizierenden Variante. Wir empfehlen deswegen, dass eine Entwicklerin unsere Betrachtungen zu zentralisierten, gleichverteilten, beschränkten Zeugen als Richtlinie für die Entwicklung von Zeugen versteht und heranzieht.

Eine Einschränkung auf nicht zentralisierte, gleichverteilte und beschränkte Zeugen ist uns jedoch auch als Richtlinie zu strikt. Das hat verschiedene Gründe, die wir im Folgenden erläutern. Es gibt nicht immer einen Zeugen, der alle Kerngedanken beachtet. Wir haben für beschränkte Zeugen bereits illustriert, dass es nicht immer einen beschränkten Zeugen gibt. Wenn ein Zeuge zusätzlich auch noch nicht zentralisiert und gleichverteilt sein soll, ergeben sich nur noch strengere Anforderungen an einen Zeugen.

Darüber hinaus kann es auch sinnvoll sein von den Kerngedanken abzuweichen. Zum Beispiel dann, wenn der ursprüngliche verteilte Algorithmus die Kerngedanken selbst nicht umsetzt. In diesem Fall kann die Berechnung eines Zeugen effizienter sein, wenn sich die Berechnung nach dem verteilten Algorithmus richtet und damit leichter in diesen integrieren lässt, als wenn sich die Berechnung nach den Kerngedanken richtet.

Ein weiterer Grund dafür, dass wir uns bei den Zeugen nicht beschränken, ist, dass wir neben den Kerngedanken auch andere Kriterien

berücksichtigen. Zum Beispiel, dass die Zeugeneigenschaft und die Verteilungseigenschaft eines Zeugenprädikats für einen Nutzer verständlich sind oder einen maschinengeprüften Beweis besitzen (siehe Teil v). Wir haben außerdem bereits einen Zusammenhang zwischen den Eigenschaften eines Zeugen auf der einen Seite und der Verteilbarkeit des entsprechenden Zeugenprädikats auf der anderen Seite thematisiert (siehe Abschnitt 6.5.4). Eine Entwicklerin einer zertifizierenden Variante eines Algorithmus sollte die verschiedenen, nicht unabhängigen Kriterien für ihren Anwendungsfall gegeneinander abwägen.

Auch bei verteilten Algorithmen gibt es neben den Kerngedanken andere Kriterien, wie eine spezielle Laufzeit, wenig Kommunikation oder sicherheitsrelevante Aspekte. Wie auch bei verteilten Algorithmen ist eine Beschränkung dann interessant, wenn wir ganz spezielle Klassen zertifizierender verteilter Algorithmen untersuchen wollen. Die zertifizierenden verteilten Algorithmen, die wir hier einführen, sollen jedoch ähnlich allgemein sein, wie die üblichen verteilten Algorithmen der Referenzliteratur [Gho14; Ray13; Lyn96; Pel00; AW04].

7.2 GETEILTE INFORMATIONEN ZWISCHEN TEILZEUGEN

In unserem einführenden Beispiel (siehe Abschnitt 6.1) ist der Zeuge so aufgebaut, dass sich einige Teilzeugen Informationen teilen. In Abschnitt 7.2.1 führen wir *überlappende* Zeugen ein; ein Zeuge ist überlappend, wenn sich einige Teilzeugen Informationen teilen. Wir erläutern außerdem, warum wir häufig überlappende Zeugen wählen. In Abschnitt 7.2.2 gehen wir darauf ein, welche Konsequenz eine verteilte Berechnung eines Zeugen für geteilte Informationen zwischen Teilzeugen hat.

7.2.1 Überlappende Zeugen

Ein verteilter Zeuge bildet *ein* Korrektheitsargument für ein Eingabe-Ausgabe-Paar. Teilzeugen liefern dabei, intuitiv gesprochen, die Puzzleteile für dieses Korrektheitsargument und geteilte Informationen zwischen den Teilzeugen sorgen dafür, dass die Puzzleteile zusammenpassen. [1]

1 Für jene Leser:innen, die sich für Kategorientheorie begeistern, ist es vielleicht interessant, dass es hier, einer Publikumsdiskussion nach auf dem NASA Formal Methods Symposium 2017 anlässlich des Artikels [VA17], einen Bezug zur Garbentheorie (engl. sheaf theory) gibt.

Wenn ein Zeuge mindestens zwei Teilzeugen mit geteilten Informationen hat, dann nennen wir diesen Zeugen einen *überlappenden* Zeugen. Wir definieren überlappende Zeugen:

Definition 27 (überlappender Zeuge)**.** *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$ *seien* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

w *ist* überlappend, *falls es* $u, v \in V$ *gibt, sodass* $W_u \cap W_v \neq \emptyset$.

Beispiel 19. *Der Zeuge, den wir in dem einführenden Beispiel des zertifizierenden Bipartitheitstests vorgestellt haben (siehe Abschnitt 6.1), ist überlappend. Für Nachbarn* u, v *gilt, dass die Variable für die Farbe von* u *sowohl in* W_u *als auch in* W_v *ist.*

Wenn ein Zeuge überlappend ist, so kann es sein, dass einer Variablen in einem Teilzeugen ein anderer Wert zugewiesen wird wie in einem anderen Teilzeugen. Die Teilzeugen widersprechen sich in diesem Fall in Informationen. Ein Zeuge, der nicht überlappend ist, hat also den Vorteil, dass es keine sich widersprechenden Informationen geben kann.

Für Eigenschaften, die für jede Komponente unabhängig vom Rest des Netzwerks gelten, lassen sich Zeugen finden, die nicht überlappen. So zum Beispiel für die Eigenschaft, ob ein Netzwerk ein eulerscher Graph ist. Ein zusammenhängender, ungerichteter Graph, bei dem jeder Knoten eine gerade Anzahl an Nachbarn hat, ist eulersch (Charakterisierung nach Euler und Hierholzer aus dem Jahr 1873).

Ein zentralisierter Zeuge, bei dem genau eine Komponente alle Informationen des Zeugen in ihrem Teilzeugen hat, ist ein Zeuge, der auf triviale Weise *nicht* überlappend ist. Schließen wir jedoch zentralisierte Zeugen aus, so gibt es nicht immer einen Zeugen, der nicht überlappend ist.

Beobachtung. *Betrachten wir zum Beispiel den Zeugen aus unserem einführenden Beispiel, dem Bipartitheitstest. Der verteilte Zeuge ist eine Bipartition des Netzwerks, wobei jeder Teilzeuge eine Bipartition einer Nachbarschaft ist. Nehmen wir als Vereinfachung an, dass nur zwei Komponenten eine Bipartition des Netzwerks als Teilzeugen unter einander aufteilen. Die beiden Teilzeuge sind dann jeweils eine Bipartition eines Teilgraphen des Netzwerks, sodass die Teilgraphen gemeinsam das gesamte Netzwerk bilden. Die beiden Teilzeugen müssen sich mindestens die Farbe einer Komponente als Information teilen. Täten sie dies nicht, dann gäbe es keinerlei Bezug zwischen den beiden Bipartitionen der Teilgraphen.*

7.2.2 Verteilte Berechnung eines Zeugen

Ein Zeuge wird von einem zertifizierenden verteilten Algorithmus so berechnet, dass jede Komponente ihren eigenen Teilzeugen berechnet.

Die Berechnung der Teilzeugen ist also Teil der nicht vertrauenswürdigen Berechnung. Ein Zeuge ist selbst also nicht vertrauenswürdig.

Bei einem überlappenden Zeugen gibt es mindestens zwei Teilzeugen, die sich Informationen teilen. Diese beiden Teilzeugen werden von den entsprechenden Komponenten unabhängig von einander berechnet. Es kann deswegen passieren, dass sich die beiden Teilzeugen in ihren geteilten Informationen widersprechen. In diesem Fall bilden die Teilzeugen kein Korrektheitsargument für ein Eingabe-Ausgabe-Paar. Ein überlappender Zeuge bringt also neue Herausforderungen mit sich.

Auch eine Eingabe oder Ausgabe wird verteilt berechnet und dennoch fordern wir nicht explizit eine Widerspruchsfreiheit zwischen beispielsweise den Teilausgaben. Der Grund dafür ist, dass die Eingabe-Ausgabe-Spezifikation in diesem Fall vorgibt, was für die geteilten Informationen gelten soll. Wir haben geteilte Informationen zwischen Teilausgaben zum Beispiel bei der Wahl eines Koordinators gesehen, wo jede Komponenten als Teilausgabe den gewählten Koordinator hat (siehe Beispiel 1 in Abschnitt 5.4.3). In diesem Fall gibt die Spezifikation zum Beispiel eine Einigkeit für die geteilte Information vor.

7.3 KONSISTENTE ZEUGEN

Wir beschäftigen uns in diesem Abschnitt mit der Konsistenz eines Zeugen. Intuitiv gesprochen meinen wir damit die Widerspruchsfreiheit aller geteilten Informationen eines überlappenden Zeugen. Einige der Resultate dieses Abschnitts haben wir in [VA18] veröffentlicht.

In Abschnitt 7.3.1 definieren wir konsistente Teilzeugen, sowie konsistente Zeugen. In Abschnitt 7.3.2 gehen wir darauf ein, dass mit der Forderung nach Konsistenz eines Zeugen eine neue Aufgabe für einen Checker entsteht: die Konsistenzprüfung.

In Abschnitt 7.3.3 führen wir zusammenhängende Zeugen ein; eine Eigenschaft eines Zeugen, die sich auf dessen Konsistenzprüfung auswirkt. In Abschnitt 7.3.4 betrachten wir dann die Konsistenz zusammenhängender Zeugen.

In Abschnitt 7.3.5 führen wir wohlgeformte Zeugen ein, die unsere Betrachtungen zur Konsistenzprüfung vereinfachen.

7.3.1 Definition konsistenter Zeugen

Wir definieren konsistente Teilzeugen, sowie konsistente Zeugen:

Definition 28 (konsistente Teilzeugen, konsistenter Zeuge)**.** *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$: W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

(i) *Für Komponenten* $u, v \in V$ *sind die Teilzeugen* $w_u \subseteq w$ *und* $w_v \subseteq w$ konsistent, *falls für alle Variablen* $a \in W_u \cap W_v$ *gilt* $w_u(a) = w_v(a)$.

(ii) w *ist* konsistent, *falls* $w \in [W]_{\mathrm{Val}_W}$.

Beispiel 20 (Zeuge ist Bipartition)**.** *Wir beziehen uns auf das Beispiel des Bipartitheitstest aus Abschnitt 6.1. In der Abbildung 6.3 auf Seite 64 sehen wir zum Beispiel, dass der Teilzeuge der Komponente 3 Variablen mit der Komponente 6 gemein hat:* $W_3 \cap W_6 = \{\mathtt{color}_3, \mathtt{color}_6\}$. *Die Teilzeugen* w_3 *und* w_6 *sind konsistent, denn es gilt* $w_3(\mathtt{color}_3) = \mathtt{black} = w_6(\mathtt{color}_3)$, *sowie* $w_3(\mathtt{color}_6) = \mathtt{white} = w_6(\mathtt{color}_6)$.

Der gesamte Zeuge ist in der Abbildung nicht explizit angegeben. Er ist jedoch implizit durch die grafische Darstellung der Bipartition des Netzwerks angegeben:

$$\begin{aligned} w = \{&(\mathtt{color}_1, \mathtt{white}),\\ &(\mathtt{color}_2, \mathtt{black}),\\ &(\mathtt{color}_3, \mathtt{black}),\\ &(\mathtt{color}_4, \mathtt{black}),\\ &(\mathtt{color}_5, \mathtt{white}),\\ &(\mathtt{color}_6, \mathtt{white})\} \end{aligned}$$

w ist konsistent, denn es gilt $w \in [W]$.

Wenn für zwei Komponenten $u, v \in V$ gilt $W_u \cap W_v \neq \emptyset$, dann haben ihre Teilzeugen gleiche Variablen; die Teilzeugen besitzen also geteilte Informationen.

Wir können die Konsistenz eines Zeugen durch seine Teilzeugen herleiten. Für einen konsistenten Zeugen gilt:

Lemma 7.3.1. *Ein potenzieller Zeuge ist genau dann konsistent, wenn seine Teilzeugen paarweise konsistent sind.*

Beweis. Sei $w \in [W]$ ein konsistenter (potenzieller) Zeuge. Dann gilt für alle Variablen $a \in W$, es gibt eine eindeutige Belegung für a mit dem Wert $w(a)$. Somit gilt für alle Komponenten $u, v \in V$ und deren Teilzeugen $w_u, w_v \subseteq w$: $w_u(a) = w(a) = w_v(a)$ für alle $a \in W_u \cap W_v$. Daraus folgt per Definition 28, dass alle Teilzeugen paarweise konsistent sind.

Für die Rückrichtung nehmen wir an, dass die Teilzeugen aller Komponenten des Zeugen $w \in [\![W]\!]$ paarweise konsistent sind. Für alle $a \in W$ gibt es mindestens eine Komponente, ohne Beschränkung der

Allgemeinheit sei diese Komponente $v \in V$, sodass $a \in W_v$, denn es gilt $W = \cup_{v \in V} W_v$. Für jede andere Komponente $u \in V$ mit $a \in W_u$ gilt dann $w_u(a) = w_v(a)$, da alle Teilzeugen paarweise konsistent sind. Folglich ist $w \in [W]$ und somit konsistent. □

7.3.2 Konsistenzprüfung als Aufgabe eines Checkers

Für einen Zeugen eins zertifizierenden verteilten Algorithmus gilt meist, dass er überlappend ist (siehe Abschnitt 7.2). Für das Korrektheitsargument, das ein Zeuge repräsentiert, ist unabdingbar, dass die Teilzeugen konsistent sind. Durch die verteilte Berechnung eines Zeugen und die Unvertrauenswürdigkeit der Komponenten, ist die Konsistenz eines berechneten Zeugen jedoch nicht garantiert. Es ist deshalb auch Aufgabe des Checkers eines zertifizierenden verteilten Algorithmus die Konsistenz eines Zeugen zu prüfen.

Für Zeugen zertifizierender sequentieller Algorithmen gibt es keine Entsprechung zu der Konsistenz eines verteilten Zeugen. Ein Checker eines zertifizierenden verteilten Algorithmus muss also im Vergleich zu einem Checker eines zertifizierenden sequentiellen Algorithmus eine zusätzliche Aufgabe übernehmen: die Prüfung der Konsistenz eines Zeugen.

Eine Folge des Lemmas 7.3.1 ist, dass ein Checker die Konsistenz eines Zeugen prüfen kann, indem er die Konsistenz der Teilzeugen paarweise prüft. Diese Art der Prüfung kann in verschiedenen Varianten verteilt umgesetzt werden. Für Zeugen mit speziellen Eigenschaften gibt es jedoch auch noch ganz andere Möglichkeiten der Konsistenzprüfung.

In Teil iv beschäftigen wir uns mit Checkern. Unter anderem stellen wir verschiedene Varianten zur Prüfung der Konsistenz vor. Die Prüfungen der Konsistenz eines Zeugen unterscheiden sich in Abhängigkeit verschiedener Anforderungen an ein Netzwerk von einander, aber auch in Abhängigkeit der Eigenschaften des zu prüfenden Zeugen. In Hinblick auf die Prüfung der Konsistenz eines Zeugen beleuchten wir im Folgenden verschiedene Eigenschaften eines Zeugen.

7.3.3 Zusammenhängende Zeugen

Eine Möglichkeit, die erlaubt bei der Konsistenzprüfung nicht alle Teilzeugen paarweise zu prüfen, ist die Einschränkung auf *zusammenhängende* Zeugen. Um einen zusammenhängenden Zeugen zu definieren, führen wir zunächst a-*Komponenten* ein.

Definition 29 (a-Komponente)**.** *Seien* N, W, *sowie für alle* $v \in V$: W_v *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

Eine Komponente $v \in V$ ist eine α-Komponente, *falls* $\alpha \in W_v$.

Beispiel 21. *In der Abbildung 6.3 auf Seite 64 sehen wir zum Beispiel, dass die Komponente 2 eine* $\texttt{color}_6$*-Komponente ist.*

Wir definieren zusammenhängende Zeugen mithilfe von α-Komponenten wie folgt:

Definition 30 (zusammenhängender Zeuge)**.** *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$: W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).*

$w \in [\![W]\!]$ *ist* zusammenhängend, *falls für alle* $\alpha \in W$ *gilt, dass der durch die* α*-Komponenten induzierte Teilgraph zusammenhängend ist.*

Beispiel 22 (Zeuge ist Bipartition)**.** *Der Zeuge aus unserem einführenden Beispiel zum Bipartitheitstest (Abschnitt 6.1) ist zusammenhängend. In der Abbildung 6.3 auf Seite 64 sehen wir zum Beispiel, dass die Komponenten 2, 3 and 6 genau die* $\texttt{color}_6$*-Komponenten sind. Der induzierte Teilgraph ist folglich zusammenhängend. Die Abbildung 7.1 zeigt den, durch die* $\texttt{color}_6$*-Komponenten induzierten, Teilgraphen. Auch für jede andere Variable* $\alpha \in W$ *gilt, dass der induzierte Teilgraph zusammenhängend ist. Somit ist der Zeuge zusammenhängend.*

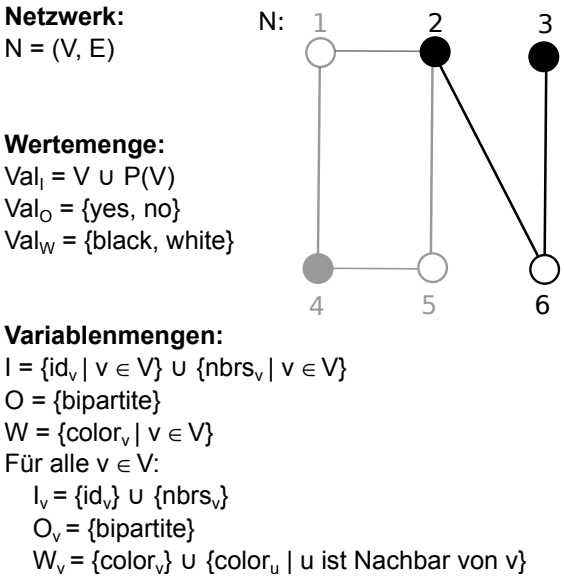


Abbildung 7.1: Die Abbildung zeigt den, durch die $\texttt{color}_6$-Komponenten induzierten, Teilgraphen in dem ansonsten ausgegrauten Netzwerk N. Der Teilgraph ist zusammenhängend.

Es gilt, dass ein beschränkter Zeuge immer auch zusammenhängend ist:

Lemma 7.3.2. *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$: W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

Wenn w beschränkt ist, dann ist w auch zusammenhängend.

Beweis. Nehmen wir also an, dass w beschränkt ist. Falls w nicht überlappend ist, gibt es nichts weiter zu zeigen. Andernfalls gibt es zwei Komponenten $u, v \in V$, sodass $W_u \cap W_v \neq \emptyset$. Dann folgt aus der Definition 26 eines beschränkten Zeugen, dass es für alle $a \in W_u \cap W_v$ einen gemeinsamen Nachbarn $x \in V$ von u und v gibt, sodass $a \in W_x$. Folglich ist w zusammenhängend. □

Andersherum gilt die Aussage nicht, denn ein zusammenhängender Zeuge ist nicht zwangsläufig auch beschränkt. Zum Beispiel kann bei einem zusammenhängenden Zeugen eine Information zwischen allen Teilzeugen eines Netzwerks geteilt werden, also ganz unabhängig von der Nachbarschaftsrelation. Bei einem beschränkten Zeugen ist eine global geteilte Information nicht möglich.

7.3.3.1 *Nicht zusammenhängende Zeugen*

Ein Zeuge muss nicht zusammenhängend sein. Wir geben zur Illustration folgend einen Zeugen an, der nicht zusammenhängend ist.

Beispiel 23 (Zeuge ist nicht zusammenhängend)**.** *Nehmen wir ein bipartites Netzwerk an, in dem Komponenten gleicher Farbe gemeinsam ein Problem lösen. Weiterhin nehmen wir an, ein Teil dieses Problems besteht darin, dass die Komponenten sich auf eine gemeinsame Auswahl aus gegebenen Optionen einigen. Dabei müssen sich jeweils die Komponenten gleicher Farbe einstimmig auf eine Auswahl einigen. Es handelt sich um eine Variante des Problems eines Konsens [Lyn96].*

Unser Ziel ist es nun, den Konsens der Komponenten gleicher Farbe zu verifizieren. Dafür hat jede Komponente einen Teilzeugen bestehend aus der eigenen Auswahl, sowie der Auswahl jeder ihrer 2-Nachbarn. Die 2-Nachbarn einer Komponente haben in einer Bipartition die gleiche Farbe wie die Komponente. Das Zeugenprädikat ist erfüllt, wenn alle Komponenten gleicher Farbe die gleiche Auswahl haben. Das Prädikat ist universell-verteilbar mit einem lokalen Prädikat, das für eine Komponenten gilt, wenn die 2-Nachbarn die gleiche Auswahl haben wie die Komponente.

Der Zeuge ist nicht zusammenhängend. Die Teilzeugen von 2-Nachbarn besitzen geteilte Variablen. Direkte Nachbarn teilen sich diese Variablen jedoch nicht in ihren Teilzeugen. Für diese Variablen ist der, durch die entsprechenden Komponenten, induzierte Teilgraph also nicht zusammenhängend.

Die Zeugen, die wir in dieser Arbeit betrachten, sind alle zusammenhängend. Das liegt zum Beispiel daran, dass wir häufig induktive Argumente für die Zeugeneigenschaft benutzen. Dabei argumentieren wir für eine Eigenschaft zum Beispiel über einen Pfad, auf dem genau benachbarte Komponenten Teilzeugen mit geteilten Informationen besitzen.

7.3.3.2 *Generalisierbarkeit des Zusammenhangs eines Zeugen*

Wir können uns auf zusammenhängende Zeugen beschränken, ohne dass dadurch die Aussagekraft eines Zeugen beeinträchtigt wird. Denn es gilt, wenn es einen Zeugen für ein Eingabe-Ausgabe-Paar gibt, dann gibt es auch einen kanonisch zusammenhängenden Zeugen für dasselbe Eingabe-Ausgabe-Paar:

Lemma 7.3.3. *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ) *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge. Sei* $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein Zeugenprädikat für* (ϕ,ψ).

Für alle Tripel (i, o, w) *gilt, wenn* $(i, o, w) \in \Gamma$, *dann existiert ein Tripel* (i, o, w'), *sodass* $(i, o, w') \in \Gamma$ *und sodass* w' *zusammenhängend ist.*

Beweis. Für den Fall,dass w bereits zusammenhängend ist, ist $w' = w$ und es gibt nichts weiteres zu zeigen.

Wir betrachten den Fall, indem $w \in [\![W]\!]$ nicht zusammenhängend ist. Dann gibt es Komponenten $u, v \in V$, sodass $a \in W_u \cap W_v$. Weiterhin gibt es keinen Pfad $p = (u, x_1, x_2, ..., x_m, v)$ zwischen u und v über Komponenten $x_l \in V$ mit $l = 1, 2, .., m$, sodass $a \in W_{x_l}$.

Wir konstruieren einen zusammenhängenden Zeugen w' auf w aufbauend. Dafür fügen wir für jedes solch beschriebene Paar aus Komponenten u, v auf einem Pfad zwischen u und v die fehlenden Variablen $a \in W_u \cap W_v$ ein. Ohne Beschränkung der Allgemeinheit sei dieser Pfad $p = (u, x_1, x_2, ..., x_m, v)$. Für jede Komponente x_l mit $l = 1, 2, ..., m$ ist dann $W'_{x_l} := W_{x_l} \cup \{a\}$. Folglich ist w' zusammenhängend.

Damit weiterhin gilt $(i, o, w') \in \Gamma$, konstruieren wir Teilzeugen w'_{x_l}, indem wir die Belegungen von u (oder analog v) übernehmen: $w'_{x_l} := w_{x_l} \cup \{(a, w_u(a)) | a \in W'_{x_l} \setminus W_{x_l}\}$ für alle $l = 1, 2, ..., m$. Da w und w' jeweils die Vereinigung der Teilzeugen ist, gilt also $w = w'$. Folglich gilt $(i, o, w') \in \Gamma$ und w' ist ein zusammenhängender Zeuge für das Eingabe-Ausgabe-Paar (i, o). □

Wir sehen also, sobald es einen Zeugen für ein Eingabe-Ausgabe-Paar gibt, gibt es auch einen zusammenhängenden Zeugen für das Eingabe-Ausgabe-Paar. Der Beweis zeigt konstruktiv, wie so ein zusammenhängender Zeuge aussieht. Wie ein Zeuge dabei konkret verteilt berechnet wird, bleibt jedoch offen. Für die verteilte Berechnung müssen die Komponenten des Netzwerks noch einen Pfad wählen.

7.3.4 Konsistenz zusammenhängender Zeugen

In diesem Abschnitt zeigen wir, dass es für die Konsistenz eines zusammenhängenden Zeugen ausreicht, wenn alle Nachbarn paarweise konsistente Teilzeugen haben.

Wir definieren Konsistenz für Nachbarschaften:

Definition 31 (konsistente Nachbarschaft)**.** *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$*:* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

$v \in V$ *hat eine* konsistente *Nachbarschaft, falls für alle Nachbarn* u *von* v *gilt, die Teilzeugen* $w_v \subseteq w$ *und* $w_u \subseteq w$ *sind konsistent.*

Für die Konsistenz eines zusammenhängenden Zeugen gilt dann folgendes:

Theorem 7.3.4. *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$*:* W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge und zusammenhängend.*

w *ist genau dann konsistent, wenn die Nachbarschaft für alle* $v \in V$ *konsistent ist.*

Beweis. Wenn w konsistent ist, dann folgt aus Lemma 7.3.1, dass alle Teilzeugen von w paarweise konsistent sind. Folglich sind auch alle Nachbarschaften konsistent.

Für die andere Richtung, nehmen wir zwei Komponenten $u, v \in V$ an, für die gilt $a \in W_u \cap W_v$. Da w zusammenhängend ist, folgt direkt aus der Definition, dass es einen Pfad zwischen u und v gibt, der ausschließlich über a-Komponenten führt. Da alle Nachbarschaften konsistent sind, haben auf diesem Pfad alle Nachbarn konsistente Teilzeugen. Durch Transitivität folgt, dass auch u und v konsistente Teilzeugen haben. Folglich ist w konsistent. □

Eine Folge des Theorems ist, dass es für Konsistenzprüfung eines zusammenhängenden Zeugen genügt die Konsistenz in der Nachbarschaft zu prüfen. Damit ist die Konsistenzprüfung für zusammenhängende Zeugen verteilt leicht umzusetzen, sehr lokal und immer gleich. Die benötigte Kommunikation ist außerdem gering und Zusammenhang ist generalisierbar für Zeugen. Für Zeugen, die nicht zusammenhängen, gilt nur die eine Richtung: wenn eine Zeuge konsistent ist, dann sind auch alle Nachbarschaften konsistent.

7.3.5 Wohlgeformte Zeugen

Häufig beinhaltet der Teilzeuge einer Komponente Informationen über die Teilausgabe einer anderen Komponente [VR15; VA17; Völ17]. Dieser Umstand wirkt sich auf die Betrachtungen zur Konsistenz aus.

7.3.5.1 *Motivation am Beispiel einer Konsensfindung*

Kommen wir auf das Problem eines Konsens zurück, wie wir es bereits in einer Variante in dem Beispiel 23 betrachtet haben. Hier sollen sich nun alle Komponenten eines Netzwerks einstimmig auf eine Auswahl einigen. Es handelt sich um eine Variante des Problems eines Konsens [Lyn96].

Sei die Teilausgabe einer Komponente die eigene Auswahl und der Teilzeuge einer Komponente jeweils die Auswahl aller Nachbarn. Die grundlegende Idee der zertifizierenden Variante ist dann, dass es im gesamten Netzwerk einen Konsens gibt, wenn die Auswahl in jeder Nachbarschaft übereinstimmt. Es handelt sich also um ein universell-verteilbares Zeugenprädikat.

Damit dieser Schluss jedoch zulässig ist, müssen die Teilzeugen wie sonst auch konsistent sein. Sie müssen darüber hinaus aber auch mit dem Ausgabewert der jeweiligen Komponente übereinstimmen. Anders formuliert muss es auch Konsistenz für die geteilten Informationen zwischen den Teilzeugen einiger Komponenten und den Teilausgaben anderer Komponenten geben.

Um zu verstehen, warum der Ausgabewert der jeweiligen Komponente mit dem entsprechenden Wert in der Teilzeugen übereinstimmen muss, stellen wir uns folgendes Szenario vor. Alle Komponenten mit Ausnahme einer Komponente v einigen sich auf eine Auswahl. Die verteilte Ausgabe ist entsprechend kein Konsens und damit nicht korrekt.

Nehmen wir nun an, dass zusätzlich ein fehlerhafter Zeuge vorliegt. Eine Information über die Auswahl von v ist Teil aller Teilzeugen der Nachbarn von v. Nehmen wir weiterhin an, die Teilzeugen der Nachbarn behaupten fälschlicherweise, dass v die gleiche Auswahl wie sie selbst hätten. Das lokale Prädikat – die Nachbarschaft stimmt in der Auswahl überein – ist folglich für alle Nachbarn von v erfüllt. Wenn nun v selbst auch einen fehlerhaften Teilzeugen besitzt, nämlich einen der behauptet, die Nachbarn hätten die gleiche Auswahl wie v getroffen, dann gilt das lokale Prädikat auch für v.

Somit ist das Zeugenprädikat erfüllt, obwohl ein inkorrektes Eingabe-Ausgabe-Paar vorliegt. Das widerspricht natürlich der Zeugeneigenschaft. Der Grund dafür ist die fehlende Konsistenz für die Teilausgaben.

7.3.5.2 *Definition: Wohlgeformte Zeugen*

Wir definieren wohlgeformte Zeugen:

Definition 32 (wohlgeformter Zeuge)**.** *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$: W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

$w \in [\![W]\!]$ *ist* wohlgeformt, *falls für alle* $u, v \in V$ *und alle* $a \in W_u$ *gilt: wenn* $a \in I_v \cup O_v$, *dann* $a \in W_v$.

Beispiel 24. *Das soeben angeführte Beispiel des Konsens hat einen wohlgeformten Zeugen, wenn für jede Komponente zusätzlich die eigene Auswahl ein Teil des eigenen Teilzeugen ist.*

Beobachtung (Prüfung der Konsistenz versus der Wohlgeformtheit)**.** *Während die Wohlgeformtheit eines Zeugen geprüft werden kann, indem wir die Variablen betrachten, müssen wir für die Konsistenz die Werte, mit denen die Variablen belegt sind, betrachten. Die Belegung der Variablen ändert sich mit jeder Ausführung. Als Folge muss die Konsistenz zur Laufzeit geprüft werden, während das für die Wohlgeformtheit nicht unbedingt nötig ist.*

Wenn der Teilzeuge einer Komponente alle Informationen der Teileingabe und der Teilausgabe enthält, ist er auch wohlgeformt:

Lemma 7.3.5. *Seien* N, W, Val_W, $[\![W]\!]$, *sowie für alle* $v \in V$: W_v *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge.*

Wenn für alle $v \in V$ *gilt* $i_v \subseteq w_v$ *und* $o_v \subseteq w_v$, *dann ist* w *wohlgeformt.*

Beweis. Wenn für alle $v \in V$ gilt $i_v \subseteq w_v$ und $o_v \subseteq w_v$, dann gilt auch für alle $v \in V$: $i_v \subseteq w$ und $o_v \subseteq w$ und somit die Wohlgeformtheit von w. □

Eine einfache, aber meist nicht so effiziente Lösung ist es also, die Teileingabe und Teilausgabe einer Komponente als Teil des Teilzeugen der Komponente aufzufassen.

7.3.5.3 *Generalisierbarkeit der Wohlgeformtheit eines Zeugen*

Wenn es einen Zeugen für ein Eingabe-Ausgabe-Paar gibt, dann gibt es auch einen kanonisch wohlgeformten Zeugen für dasselbe Eingabe-Ausgabe-Paar:

Lemma 7.3.6. *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ) *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $w \in [\![W]\!]$ *ein potenzieller Zeuge. Sei* $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein Zeugenprädikat für* (ϕ,ψ).

Für alle Tripel (i, o, w) *gilt, wenn* $(i, o, w) \in \Gamma$, *dann existiert ein Tripel* (i, o, w'), *sodass* $(i, o, w') \in \Gamma$ *und* w' *wohlgeformt ist.*

Beweis. Wir können w' immer so wählen, dass wir w, wie in Lemma 7.3.5 gezeigt, um die Eingabe und Ausgabe erweitern. □

Wir können uns folglich auf wohlgeformte Zeugen beschränken, ohne dass dadurch die Aussagekraft eines Zeugen beeinträchtigt wird.

7.3.5.4 *Bedeutung der Wohlgeformtheit für die Konsistenzprüfung*

Die Beschränkung auf wohlgeformte Zeugen hat den Vorteil, dass wir die Betrachtungen zur Konsistenz einfacher halten, indem wir nur Zeugen und nicht zusätzlich noch Eingabe und Ausgabe betrachten. Das macht unsere Betrachtungen hier auf Papier erst einmal übersichtlicher.

Für eine Implementierung einer Konsistenzprüfung kann eine andere Lösung sowohl algorithmisch als programmiertechnisch effizienter sein. Ist ein Zeuge nicht wohlgeformt, so müssen im Zweifelsfall eben auch die entsprechenden Teile der Eingabe und Ausgabe in die Prüfung der Konsistenz einbezogen werden. Unsere Ergebnisse zur Konsistenz eines Zeugen sind für diesen Fall auch direkt übertragbar.

Eine Betrachtung zur Wohlgeformtheit eines Zeugen ist jedoch selbst dann interessant, wenn ein wohlgeformter Zeuge nicht das Ziel ist. Denn bei der Wohlgeformtheit geht es im Grunde um die Frage, welche Informationen geteilt werden und entsprechend auf Konsistenz geprüft werden müssen. Bei der Entwicklung einer zertifizierenden Variante eines verteilten Algorithmus ist es deswegen sinnvoll über Wohlgeformtheit nachzudenken.

Während die Wohlgeformtheit eines Zeugen geprüft werden kann, indem wir die Variablen betrachten, müssen wir für die Konsistenz die Werte, mit denen die Variablen belegt sind, betrachten. Für die Wohlgeformtheit spielt deswegen auch eine Rolle, wie die Variablen im Netzwerk repräsentiert sind. Die Wohlgeformtheit könnte beispielsweise mit der korrekten Initialisierung eines Netzwerks garantiert werden. Die Belegung der Variablen wiederum ändert sich mit jeder Ausführung. Als Folge muss die Konsistenz zur Laufzeit geprüft werden, während das für die Wohlgeformtheit nicht unbedingt nötig ist.

8 ZERTIFIZIERENDE VERTEILTE ALGORITHMEN

Wir fügen in diesem Kapitel nun die Konzepte zusammen, die wir in den vorigen Kapiteln dieses Teils eingeführt haben. Wir definieren somit zertifizierende verteilte Algorithmen, die

- terminieren,
- ihr verteiltes Eingabe-Ausgabe-Paar verifizieren,
- ein verteilbares Zeugenprädikat besitzen und
- verteilte Zeugen berechnen.

In Abschnitt 8.1 definieren wir zertifizierende verteilte Algorithmen. In Abschnitt 8.2 zeigen wir, wie die Instanzverifikation mithilfe eines zertifizierenden verteilten Algorithmus gelingt.

In Abschnitt 8.3 zeigen wir, dass ein zertifizierender verteilter Algorithmus ein Spezialfall eines zertifizierenden sequentiellen Algorithmus ist und darüber hinaus, dass es für jeden korrekten verteilten Algorithmus eine zertifizierende verteilte Variante gibt.

In Abschnitt 8.4 widmen wir uns der Frage, was eine „gute“ zertifizierende Variante verteilter Algorithmen ausmacht.

Für eine bessere Lesbarkeit nehmen wir in diesem Kapitel die Objekte eines festen, aber beliebigen Interfaces eines zertifizierenden verteilten Algorithmus an, wie bereits in den vorigen Kapiteln. Zur Erinnerung siehe Abschnitt 6.2.3 auf Seite 66. Dort haben wir die Objekte mit ihrer Bedeutung zum einfachen Nachschlagen in der Abbildung 6.4 aufgelistet.

8.1 DEFINITION ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Wir definieren zertifizierende verteilte Algorithmen, wie folgt:

Definition 33 ((vollständig) zertifizierender verteilter Algorithmus)**.** *Seien* N, I, O, W, Val_I, Val_O, Val_W, $[\![I]\!]$, $[\![O]\!]$, $[\![W]\!]$ *und* (ϕ, ψ), *sowie für alle* $v \in V$ *seien* I_v, O_v, W_v, $[I_v]$, $[O_v]$ *und* $[W_v]$ *wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66). Sei* $\Gamma \subseteq [\![I]\!] \times [\![O]\!] \times [\![W]\!]$ *ein verteilbares Zeugenprädikat für* (ϕ, ψ) *in* N. *Sei* A *ein verteilter Algorithmus, sodass* A *für eine Eingabe* i *die Ausgabe* o *und den potenziellen Zeugen* w *berechnet,*

indem jede Komponente $v \in V$ *für eine Teileingabe* $i_v \in [I_v]$*, sowohl eine Teilausgabe* $o_v \in [O_v]$ *als auch einen Teilzeugen* $w_v \in [W_v]$ *berechnet.*

(i) A *mit* Γ *ist ein* zertifizierender *verteilter Algorithmus für* (ϕ, ψ)*.*

(ii) A *mit* Γ *ist ein* vollständig *zertifizierender verteilter Algorithmus für* (ϕ, ψ) *falls gilt:*

(a) A *mit* Γ *ist ein korrekter (terminierender) verteilter Algorithmus für ein durch* (ϕ, ψ) *spezifiziertes Problem,*

(b) Γ *ist ein vollständiges Zeugenprädikat für* (ϕ, ψ)*,*

(c) Γ *ist vollständig verteilbar in* N*,*

(d) Γ *ist entscheidbar*

und für alle i, o, w *gilt:*

(e) $(i, o, w) \in \Gamma$*,*

(f) w *ist wohlgeformt und*

(g) w *ist konsistent.*

Beispiel 25. *Wir haben für die einzelnen Konzepte, die wir in den Bedingungen fordern, in den vorigen Kapiteln bereits jeweils Beispiele gesehen. Für ein umfassendes Beispiel eines zertifizierenden verteilten Algorithmus verweisen wir auf die Fallstudien, die wir im folgenden Kapitel 9 vorstellen.*

Die Definition eines zertifizierenden verteilten Algorithmus ist so gewählt, dass sie aus der Sicht eines Nutzers im Sinne der Instanzverifikation ausreichend ist.

Hingegen ist die Definition eines vollständig zertifizierenden verteilten Algorithmus so gewählt, dass sie der Sicht einer Entwicklerin eines solchen Algorithmus entspricht. Ein vollständig zertifizierender Algorithmus verifiziert jedes seiner Eingabe-Ausgabe-Paare zur Laufzeit. Die Bedingung (a) garantiert, dass der Algorithmus nur korrekte Eingabe-Ausgabe-Paare berechnet, die Bedingungen (b) und (c) garantieren, dass es immer einen verteilten Zeugen für ein korrektes Eingabe-Ausgabe gibt, die Bedingung (d), dass es einen Checker gibt und die Bedingungen (e)–(f), dass der Algorithmus immer einen Zeugen zur Verifikation berechnet.

Wir erläutern im folgenden Abschnitt die Instanzverifikation aus der Sicht eines Nutzers und der Sicht einer Entwicklerin.

8.2 INSTANZVERIFIKATION FÜR VERTEILTE ALGORITHMEN

In Abschnitt 8.2.1 formulieren wir das Instanzverifikationsproblem für *verteilte* Eingabe-Ausgabe-Paare analog zum Instanzverifikations-

problem für Eingabe-Ausgabe-Paare sequentieller Algorithmen (siehe Abschnitt 3.1 auf Seite 21).

In Abschnitt 8.2.2 beweisen wir die Instanzverifikation aus der Sicht eines Nutzers und in Abschnitt 8.2.3 aus der Sicht einer Entwicklern.

8.2.1 Instanzverifikationsproblem für verteilte Algorithmen

Das Instanzverifikationsproblem für Eingabe-Ausgabe-Paare *verteilter* Algorithmen stellt nun die Frage, ob ein *verteiltes* Eingabe-Ausgabe-Paar eine gegebene Eingabe-Ausgabe-Spezifikation erfüllt. Wir formulieren das Instanzverifikationsproblem, wie in der Abbildung 8.1 dargestellt.

Instanzverifikationsproblem:

Gegeben:
Netzwerk $N = (V, E)$
Endliche Variablenmengen I, O
Wertemengen Val_I, Val_O
Eingaben $[\![I]\!]$
Ausgaben $[\![O]\!]$
Eingabe-Ausgabe-Spezifikation $\phi \subseteq [\![I]\!]$, $\psi \subseteq [\![I]\!] \times [\![O]\!]$
Eingabe-Ausgabe-Paar (i, o) mit $i \in [\![I]\!]$, $o \in [\![O]\!]$

Frage:
Gilt $\psi(i, o) \vee \neg\phi(i)$?

Abbildung 8.1: Instanzverifikationsproblem für verteilte Eingabe-Ausgabe-Paare. Dabei seien N, I, O, Val_I, Val_O, $[\![I]\!]$, $[\![O]\!]$ und (ϕ, ψ) wie üblich definiert (siehe Abschnitt 6.2.3 auf Seite 66).

8.2.2 Instanzverifikation aus Sicht eines Nutzers

Für einen Nutzer reicht es, wenn das Instanzverifikationsproblem für genau das Eingabe-Ausgabe-Paar positiv entschieden ist, an dem er interessiert ist. Das folgende Theorem formuliert die Idee der Instanzverifikation *eines* Eingabe-Ausgabe-Paares mithilfe eines zertifizierenden verteilten Algorithmus.

Theorem 8.2.1 (Instanzverifikation I)**.** *Sei A ein zertifizierender verteilter Algorithmus für eine Eingabe-Ausgabe-Spezifikation (ϕ, ψ) mit dem verteilbaren Zeugenprädikat Γ. Wenn A eine Ausgabe o und einen potenziellen Zeugen w für eine Eingabe i berechnet, dann gilt:*

Wenn w wohlgeformt und konsistent ist und das Tripel $(i, o, w) \in \Gamma$, dann ist das Instanzverifikationsproblem für (i, o) entschieden mit $(i, o) \in \psi$ oder $i \notin \phi$.

Beweis. Folgt aus der Definition eines zertifizierenden verteilten Algorithmus und insbesondere aus der Definition 22 eines verteilbaren Prädikats und aus der Definition 15 eines Zeugenprädikats. □

Entscheidet ein Nutzer für sein Eingabe-Ausgabe-Paar also, dass der Zeuge sowohl wohlgeformt als auch konsistent ist und das verteilbare Zeugenprädikat erfüllt, so entscheidet er das Instanzverifikationsproblem für sein Eingabe-Ausgabe-Paar positiv. Diese Entscheidung ist deswegen Aufgabe eines Checkers (siehe Teil iv). Ein Nutzer muss entsprechend auf die Korrektheit eines Checkers vertrauen können (siehe Teil v). Darüber hinaus muss er auch auf die Korrektheit der Verteilungseigenschaft und der Zeugeneigenschaft des Zeugenprädikats vertrauen (siehe auch hierfür Teil v).

Beobachtung (Korrektheit & keine Vollständigkeit)**.** *Ein zertifizierender verteilter Algorithmus garantiert uns Korrektheit, aber keine Vollständigkeit. Das heißt, wenn das Instanzverifikationsproblem im Sinne des Theorems 8.2.1 für ein Eingabe-Ausgabe-Paar positiv entschieden ist, dann gibt es keinen Zweifel an der Korrektheit. Für einen zertifizierenden verteilten Algorithmus ist aber nicht garantiert, dass das Instanzverifikationsproblem für alle Eingabe-Ausgabe-Paare positiv entschieden wird. Darüber hinaus ist für Fälle, in denen es nicht positiv entschieden wird, auch kein Rückschluss möglich, dass es sich dann um ein inkorrektes Eingabe-Ausgabe-Paar handelt.*

8.2.3 Instanzverifikation aus Sicht einer Entwicklerin

Eine Entwicklerin eines verteilten Algorithmus entwickelt ihren Algorithmus so, dass er immer eine korrekte Ausgabe zu einer Eingabe berechnet. Eine zertifizierende Variante entwickelt sie außerdem so, dass ihr Algorithmus auch immer einen Zeugen für dieses Eingabe-Ausgabe-Paar berechnet. Aus der Sicht einer Entwicklerin muss das Instanzverifikationsproblem also für alle korrekten Eingabe-Ausgabe-Paare positiv entscheidbar sein. Das folgende Theorem formuliert die Idee der Instanzverifikation *für alle* Eingabe-Ausgabe-Paare mithilfe eines korrekten zertifizierenden verteilten Algorithmus.

Theorem 8.2.2 (Instanzverifikation II)**.** *Ein vollständig zertifizierender verteilter Algorithmus für eine gegebene Spezifikation, entscheidet das Instanzverifikationsproblem für all seine Eingabe-Ausgabe-Paare positiv.*

Beweis. Folgt aus der Definition 33 eines vollständig zertifizierenden verteilten Algorithmus. □

8.3 UNIVERSALITÄT ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Wir beschäftigen uns in diesem Abschnitt mit der Universalität zertifizierender verteilter Algorithmen. In Abschnitt 8.3.1 erläutern wir, warum ein zertifizierender verteilter Algorithmen ein Spezialfall eines zertifizierenden sequentiellen Algorithmen ist. In Abschnitt 8.3.2 zeigen wir, dass es für einen korrekten verteilten Algorithmus immer auch eine zertifizierende verteilte Variante gibt.

8.3.1 Spezialfall zertifizierender sequentieller Algorithmen

Theorem 8.3.1. *Ein zertifizierender verteilter Algorithmus ist ein Spezialfall eines zertifizierenden sequentiellen Algorithmus.*

Beweis. Ein zertifizierender verteilter Algorithmus berechnet eine Ausgabe und einen Zeugen für eine Eingabe, genau ein wie ein zertifizierender sequentieller Algorithmus, nur zusätzlich in verteilter Form von Teilausgaben und Teilzeugen.

Ein zertifizierender verteilter Algorithmus hat genau wie ein zertifizierender sequentieller Algorithmus ein Zeugenprädikat. Für das zugehörige Zeugenprädikat muss zusätzlich gelten, dass es verteilbar ist. □

8.3.2 Universalität

Für einen korrekten (sequentiellen) Algorithmus gibt es immer eine zertifizierende Variante [McC+11, Kapitel 5]. Wir zeigen darüber hinaus, dass es dann auch immer einen vollständig zertifizierenden verteilten Algorithmus gibt.

Theorem 8.3.2 (Universalität)**.** *Sei A ein korrekter verteilter Algorithmus für eine Spezifikation (ϕ, ψ). Es gibt einen vollständig zertifizierenden verteilten Algorithmus für (ϕ, ψ).*

Beweis. Wir nehmen einen korrekten verteilten Algorithmus an und beschreiben darauf aufbauend eine generische Konstruktion eines korrekten zertifizierenden verteilten Algorithmus.

Zeugenberechnung. Jede Komponenten sammelt als Teilzeugen die Historie ihrer lokalen Berechnungen, sowie die Historie ihrer Kommunikation.

Wohlgeformte Zeugen. Wenn der Teilzeuge einer Komponente Informationen aus einer Teileingabe oder Teilausgabe einer anderen Komponente enthält, dann muss diese Information kom-

muniziert worden sein. Damit ist die Information auch Teil der Teilzeugen all jener Komponenten, über die die Information kommuniziert wurde.

Konsistente Zeugen. Da jede Komponente nach dem korrekten verteilten Algorithmus agiert, sind die Teilzeugen paarweise konsistent und somit auch der Zeuge.

Zeugenprädikat. Das Zeugenprädikat ist erfüllt, falls der Zeuge (die gesamte Berechnung und Kommunikation) und die Ausgabe zum korrekten verteilten Algorithmus bei entsprechender Eingabe passen. Die Zeugeneigenschaft des Zeugenprädikats beruht auf dem Beweis der Korrektheit des korrekten verteilten Algorithmus.

Vollständiges Zeugenprädikat. Da der verteilte Algorithmus korrekt ist, gibt es für jedes korrekte Eingabe-Ausgabe-Paar auch eine passende Berechnung und Kommunikation.

Verteilbares Zeugenprädikat. Das Zeugenprädikat ist universell-verteilbar mit einem lokalen Prädikat, das je Komponenten erfüllt ist, falls die Berechnung und Kommunikation der Komponente korrekt ist.

Vollständig verteilbares Zeugenprädikat. Da der Algorithmus korrekt ist, gibt es für jedes korrekte Eingabe-Ausgabe-Paar auch eine passende Berechnung und Kommunikation und da er verteilt ist, gibt es gibt es für jedes korrekte Teileingabe-Teilausgabe-Paar eine passende Berechnung und Kommunikation pro Komponente.

Mit der Konstruktion erhalten wir also einen korrekten zertifizierenden verteilten Algorithmus. □

Die Konstruktion birgt die gleichen Probleme, aber auch Potenziale, wie die entsprechende Konstruktion eines zertifizierenden *sequentiellen* Algorithmus. Wir zitieren aus [McC+11, S. 30]:

> We admit that the construction above leaves much to be desired. [...] However, the construction is also quite assuring and gives strong moral support. So, when searching for a certifying algorithm we only have to try hard enough; we are guaranteed to succeed. The construction also captures the intuition that certification is no harder than a formal correctness proof of a program.

Die Konstruktion lässt deswegen zu wünschen übrig, weil wir bereits von einem korrekten verteilten Algorithmus ausgehen und damit bereits von einem Korrektheitsbeweis für den Algorithmus. Dadurch, dass dieser Korrektheitsbeweis die Grundlage für die Zeugeneigenschaft ist, reduzieren wir Zertifizierung auf vollständige formale Verifikation. Die Konstruktion bietet uns im Allgemeinen keine wün-

schenswerte zertifizierende Variante. Wir beschäftigen uns deswegen im folgenden Abschnitt mit der Güte einer Zertifizierung.

8.4 GÜTE EINER ZERTIFIZIERENDEN VARIANTE

Wir beschäftigen uns in diesem Abschnitt mit der Güte einer zertifizierenden Variante eines verteilten Algorithmus. Einige Kriterien lassen sich, wie schon bei zertifizierenden sequentiellen Algorithmen, nicht mit mathematischer Exaktheit fassen. Lassen wir die Kriterien mathematisch nicht exakt, so ist die Frage nach einer guten Zertifizierung eben auch ein bisschen Geschmackssache. Einigen wir uns jedoch ausschließlich auf formale Kriterien, so wird das Konzept, unserer Ansicht nach, zu stark eingeschränkt. Wir halten es deswegen so, wie für zertifizierende sequentielle Algorithmen; wir diskutieren Kriterien und erachten sie als wichtigen Teil des Konzepts eines zertifizierenden verteilten Algorithmus, nicht aber als Teil dessen Definition.

In Abschnitt 8.4.1 stellen wir zunächst nicht erstrebenswerte Extremfälle der Zertifizierung vor, um zu zeigen, was ausgeschlossen werden soll.

In Abschnitt 8.4.2 stellen wir formale und informelle Kriterien für die Güte von Zeugenprädikaten vor.

8.4.1 Extremfälle

Wir betrachten zwei Extremfälle, die bereits durch zertifizierende sequentielle Algorithmen bekannt sind.

Den einen Extremfall haben wir bereits kennengelernt durch die generische Konstruktion eines zertifizierenden verteilten Algorithmus, wie wir sie im Beweis des Theorems 8.3.2 zur Universalität gesehen haben. In diesem Fall wird Zertifizierung auf formale Verifikation reduziert. Die Zeugeneigenschaft benötigt einen Korrektheitsbeweis des verteilten Algorithmus beziehungsweise dessen Implementierung. In diesem Fall hat das Zeugenprädikat eine im allgemeinen komplizierte Zeugeneigenschaft.

Der andere Extremfall ist es, das Zeugenprädikat als die Nachbedingung der Eingabe-Ausgabe-Spezifikation zu wählen. In diesem Fall gibt es keinen Zeugen. In diesem Fall ist das Zeugenprädikat im allgemeinen schwer zu entscheiden.

Die Herausforderung ist es also, ein „gutes“ Zeugenprädikat zu finden.

8.4.2 Güte eines Zeugenprädikats

Wir diskutieren in diesem Abschnitt die Güte eines Zeugenprädikats, indem wir auf den Kriterien, die bereits für sequentielle Algorithmen bekannt sind [McC+11], aufbauen und um die Besonderheiten zertifizierender verteilter Algorithmen erweitern. Die beiden aufgezeigten Extremfälle verdeutlichen, was für ein Zeugenprädikat erstrebenswert ist. Zum einen soll ein Zeugenprädikat einen *einfachen* Beweis für seine Zeugeneigenschaft besitzen. Zum anderen soll ein Zeugenprädikat leicht algorithmisch *prüfbar* sein.

8.4.2.1 *Einfachheit der Zeugeneigenschaft*

Wir können nicht formalisieren, wann der Beweis der Zeugeneigenschaft einfach, also leicht verständlich ist. Es ist klar, dass dies eine subjektive Empfindung eines Nutzers festlegt. Hilfreich könnte es aber beispielsweise sein, wenn die Zeugeneigenschaft aus einem wohlbekannten Theorem folgt. Dann müsste ein Nutzer nur auf bereits etablierte Mathematik vertrauen muss. Der Beweis könnte jedoch auch als maschinen-geprüfter Beweis vorliegen. Der Nutzer müsste dann nur einem Beweisassistenten vertrauen.

8.4.2.2 *Erweiterung: Einfachheit der Verteilungseigenschaft*

Bei einem zertifizierenden verteilten Algorithmus haben wir ein verteilbares Zeugenprädikat. Für verteilbare Zeugenprädikate fordern wir zusätzlich, dass auch die Verteilungseigenschaft einen einfachen Beweis hat.

Einige Leser:innen haben vielleicht bereits bemerkt, dass wir verteilbare Zeugenprädikate auch so definieren könnten, dass wir nur eine Eigenschaft fordern, die dann die Zeugeneigenschaft und die Verteilungseigenschaft gemeinsam ausdrückt. Wir haben uns dafür entschieden beide Eigenschaften getrennt von einander aufzuführen, um die Verständlichkeit zu erhöhen.

8.4.2.3 *Prüfbarkeit des Zeugenprädikats*

Mögliche Kriterien dafür, dass es leicht ist ein Zeugenprädikat zu entscheiden, sind in [McC+11] folgende:

- Der Checker-Algorithmus hat eine geringe Laufzeit, idealerweise linear in der Größe seiner Eingabe.
- Das Zeugenprädikat hat eine einfache logische Struktur.
- Das Zeugenprädikat kann in einem einfachen Kalkül entschieden werden.

- Die Korrektheit des Checker-Programms ist offensichtlich.
- Das Checker-Programm ist formal verifiziert.

8.4.2.4 *Erweiterung: Verteilte Prüfbarkeit*

Für ein verteilbares Zeugenprädikat und einen entsprechenden verteilten Checker lassen sich alle Kriterien bis auf das Kriterium zur linearen Laufzeit direkt übertragen. Für verteilte Algorithmen können wir zwar auch die Laufzeit betrachten, üblicher ist es hierbei jedoch die Komplexität der Kommunikation zu betrachten. Wir können das Kriterium also austauschen beziehungsweise erweitern darum, dass die Kommunikation jedes Teilcheckers lokal ist.

Außerdem ist *Invasivität* (aus dem Englischen übersetzt von intrusiveness) ein Kriterium, das aus der Community der Laufzeitverifikation verteilter Systeme bekannt ist. Hierbei geht es darum, wie invasiv ein verteilter Checker in das verteilte System eingreift, also zum Beispiel Ressourcen verbraucht. Dabei wird möglichst wenig Invasivität angestrebt. Wir kommen auf die Invasivität eines verteilten Checkers im Teil iv zu verteilten Checkern zurück.

9 FALLSTUDIEN

Wir stellen in diesem Kapitel einige Fallstudien zertifizierender verteilter Algorithmen vor, um den Leser:innen einen besseren Eindruck davon zu vermitteln, wie die Zertifizierung verteilter Algorithmen gelingt. In Abschnitt 9.1 diskutieren wir unsere Auswahl an Fallstudien und erläutern ihren Aufbau.

In den Abschnitten 9.2 – 9.3 stellen wir dann die folgenden Fallstudien vor: Berechnung der kürzesten Pfade (Abschnitt 9.2), Konstruktion eines Spannbaums (Abschnitt 9.3), Breitensuche (Abschnitt 9.4), Konsensfindung (Abschnitt 9.5), Broadcast (Abschnitt 9.6), Leader-Election (Abschnitt 9.7) und Bipartitheitstest (Abschnitt 9.8). Dabei ist zu beachten, dass die Fallstudien teilweise auf den ersten beiden Fallstudien in Abschnitt 9.2 und in Abschnitt 9.3 aufbauen.

9.1 AUSWAHL UND AUFBAU DER FALLSTUDIEN

Wir diskutieren unsere Auswahl an Fallstudien (Abschnitt 9.1.1) und erläutern den Aufbau, den wir bei jeder der Fallstudien verfolgen (Abschnitt 9.1.2).

9.1.1 Auswahl

Unsere Auswahl der Fallstudien ist durch zwei Kriterien motiviert.

9.1.1.1 *Grundlegende verteilte Algorithmen*

Wir geben zertifizierende Varianten für verteilte Algorithmen an, die man üblicherweise in den Referenzwerken verteilter Algorithmen findet [AW04; Tel94; Ray13; Peloo; Lyn96]. Hierbei ist es uns wichtig, nicht nur verteilte Algorithmen einer Kategorie zu betrachten, also zum Beispiel nur diverse Algorithmen zur Konsensfindung. Unser Ziel ist es, möglichst durch eine Vielfalt an verteilten Algorithmen repräsentative Fallstudien vorzustellen. So betrachten wir beispielsweise mit Konsensfindung, Bipartitheitstest, Broadcast oder auch Leader Election unterschiedliche verteilte Algorithmen und decken damit klassische Kategorien verteilter Algorithmen ab.

9.1.1.2 *Zertifizierende Graphalgorithmen*

Die Topologie eines Netzwerks ist unmittelbar als Graph repräsentierbar und so gibt es viele verteilte Graphalgorithmen – also Algorithmen, die Probleme auf Graphen lösen. Beispiele für verteilte Graphalgorithmen sind die verteilte Berechnung kürzester Pfade, verteilter Bipartitheitstest oder verteilte Breitensuche. Erciyes widmet verteilten Graphalgorithmen sogar ein Grundlagenbuch zu verteilten Algorithmen [Erc13]. Wir finden sie als Kategorie aber auch in allen gängigen Referenzwerken verteilter Algorithmen wieder [AW04; Tel94; Ray13; Pel00; Lyn96].

Zu einem verteilten Graphalgorithmus existiert gegebenenfalls ein entsprechender zertifizierender *sequentieller* Graphalgorithmus. Aus der Literatur kennen wir bereits einige zertifizierende sequentielle Graphalgorithmen [McC+11; Sch12; BHS09; CDH13; KN09; HC11; HH05; NP12; HK07; ZPK14].

Ein verteilter Graphalgorithmus für den bereits eine zertifizierende sequentielle Variante existiert, ist besonders interessant für uns. Schließlich eignet sich eine zertifizierende verteilte Variante für einen Vergleich zwischen zertifizierenden verteilten und zertifizierenden sequentiellen Algorithmen.

Wir verfolgen deswegen bei der Entwicklung einer zertifizierenden Variante eines verteilten Graphalgorithmus das Ziel, zunächst möglichst viel von der Zertifizierung des zertifizierenden sequentiellen Algorithmus zu übernehmen und die Zertifizierung dann auf die Verteiltheit hin anzupassen.

Hierbei möchten wir jedoch auch auf einen grundsätzlichen Unterschied zwischen verteilten und sequentiellen Graphalgorithmen hinweisen. Ein sequentieller Graphalgorithmus erhält einen beliebigen Graph als Eingabe, während ein verteilter Graphalgorithmus die Topologie des zugrunde liegenden Netzwerks als Eingabegraph erhält.

9.1.2 Aufbau

Wir erläutern den Aufbau der Fallstudien, bei denen wir die Sicht eines Nutzers einnehmen.

9.1.2.1 *Instanzverifikation aus Sicht eines Nutzers*

In Kapitel 8 haben wir in Theorem 8.2.1 die Instanzverifikation mithilfe eines zertifizierenden verteilten Algorithmus aus der Sicht eines Nutzers formuliert.

Wir betrachten alle Fallstudien zertifizierender verteilter Algorithmen aus Sicht eines Nutzers. Wir geben also je Fallstudie ein verteilba-

res Zeugenprädikat an und zeigen dessen Zeugeneigenschaft, sowie dessen Verteilungseigenschaft. Wir besprechen die Entscheidung des verteilbaren Zeugenprädikats und die Berechnung der Zeugen.

Aus Sicht eines Nutzers bedeutet auch, dass wir darauf verzichten zu zeigen, dass die vorgestellten verteilten Algorithmen hier alle *vollständig* zertifizierend sind.

Der „Charme“ der Instanzverifikation mit zertifizierenden verteilten Algorithmen ist unserer Ansicht nach aus der Sicht eines Nutzers gut erkennbar. Schließlich ist die Idee eines zertifizierenden Algorithmus, dass dessen Nutzer von der Korrektheit seines Eingabe-Ausgabe-Paars überzeugt wird. Ein verteilbares Zeugenprädikat soll deswegen eine leicht verständliche Zeugeneigenschaft, sowie Verteilungseigenschaft haben. Wir denken, dass die Fallstudien dieser Anforderung genügen. Die Leser:innen müssen dies selbst beurteilen.

Festzuhalten ist jedoch, dass die Vollständigkeit eines Zeugenprädikats keinen leicht verständlichen Beweis haben muss, denn für einen Nutzer ist dessen Verständnis irrelevant. Dennoch sind die hier angegebenen zertifizierenden verteilten Algorithmen auch alle vollständig zertifizierend. Wir werden dafür jedoch lediglich hier und da eine Intuition dafür geben und auf technische Details verzichten.

Obwohl wir die Fallstudien für die Instanzverifikation aus Sicht eines Nutzers beschreiben, skizzieren wir wie die Berechnung der Zeugen aussieht. Wir möchten dabei illustrieren, dass die Integration der Berechnung der Zeugen in den ursprünglichen verteilten Algorithmus meist ohne zusätzlichen Aufwand gelingt. In den meisten Fällen müssen für die Zeugen ein paar Informationen gesammelt werden, die ohnehin während der Ausführung des jeweiligen verteilten Algorithmus anfallen.

9.1.2.2 *Eingabe und berechnete Ausgabe*

Ein verteilter Algorithmus berechnet für eine Eingabe eine Ausgabe. Zum Beispiel berechnet eine Spannbaumkonstruktion für die Eingabe eines Netzwerks als Ausgabe einen Spannbaum in dem Netzwerk. Dies ist natürlich nur der Fall, wenn der verteilte Algorithmus das Spannbaumproblem korrekt löst.

Für die Zertifizierung erachten wir den verteilten Algorithmus als nicht vertrauenswürdig. Wir wissen durch die Spezifikation des Problems zwar, dass die Ausgabe ein Spannbaum in dem Netzwerk sein soll, wir wissen jedoch nicht, ob es sich bei der berechneten Ausgabe tatsächlich um einen Spannbaum handelt.

Exakterweise müssten wir in den Fallstudien also immer von der berechneten Ausgabe sprechen. Schließlich wissen wir nur dann, dass die berechnete Ausgabe korrekt ist, wenn sie die durch das verteilbare

Zeugenprädikat geforderten Eigenschaft erfüllt. Wir erachten diese Schreibweise jedoch als verwirrend und weniger intuitiv, da man sich stets merken muss, was die korrekte Ausgabe sein soll.

Wir sprechen deswegen zum Beispiel bei der Spannbaumkonstruktion von dem *berechneten* Spannbaum. Damit ist dann immer gemeint, dass der berechnete Spannbaum gegebenenfalls gar kein Spannbaum ist. Nämlich dann nicht, wenn der verteilte Algorithmus fehlerhaft ist.

9.1.2.3 *Verteilter Checker*

Zu einem zertifizierenden verteilten Algorithmus gehört sinnvoller Weise ein verteilter Checker, der das verteilbare Zeugenprädikat entscheidet. Wir haben verteilbare Prädikate so definiert, dass sich deren Entscheidung im Wesentlichen aus der Summe der Entscheidungen der entsprechenden lokalen Prädikate ergibt. Der Teil eines verteilten Checkers der spezifisch für den konkreten zertifizierenden verteilten Algorithmus ist, ist die Entscheidung der lokalen Prädikate.

Wir betrachten in den Fallstudien deswegen auch nur, was ein Teilchecker einer Komponente dafür tun muss. Im folgenden Teil iv betrachten wir dann verteilte Checker im Allgemeinen.

9.1.2.4 *Zeugen*

Die gewählten Zeugen der Fallstudien sind alle wohlgeformt und zusammenhängend, sodass eine lokale Konsistenzprüfung möglich ist (siehe Teil iv).

Darüber hinaus fordern wir Leser:innen dazu auf, unsere Wahl der Zeugen hinsichtlich der Umsetzung der Kerngedanken der Gleichheit, Verteiltheit und Lokalität zu bewerten. Alle Zeugen sind gleichverteilt und nicht zentralisiert.

Nicht alle Zeugen sind beschränkt. Dabei ist zu beobachten, dass Zeugen entweder 2-beschränkt sind oder es kein k für die Beschränkung gibt, weil eine Information global geteilt werden muss. Am Beispiel der Zertifizierung eines Spannbaums erläutern wir, warum es keinen beschränkten Zeugen geben kann. Alle Fallstudien, die einen Spannbaum benutzen, haben folglich keine beschränkten Zeugen.

Dennoch gilt auch bei den Fallstudien mit Zeugen, die nicht beschränkt sind, dass wir den Kerngedanken der Lokalität so weit wie möglich beachtet haben. Eine Bewertung hierfür überlassen wir den Leser:innen.

9.2 KÜRZESTE PFADE

Die verteilte Berechnung der kürzesten Pfade in einem Netzwerk ist die Grundlage vieler Routingprotokolle [Med10]. Es handelt sich also um einen Grundbaustein verteilter Algorithmen [AW04; Tel94; Ray13; Peloo; Lyn96; Gho14; Erc13]. Aufgrund unseres aufgeführten Kriteriums, grundlegende verteilte Algorithmen zu zertifizieren, ist eine Zertifizierung für uns interessant.

Eine Zertifizierung ist aber auch deshalb interessant, weil es bereits einen zertifizierenden sequentiellen Algorithmus zu Berechnung der kürzesten Pfade in einem Graphen gibt [McC+11]. Hier trifft also auch unser zweites Kriterium zu. Anhand der zertifizierenden sequentiellen und der zertifizierenden verteilten Variante sind Gemeinsamkeiten und Unterschiede besonders gut erkennbar. Wir bauen deswegen folgend auf der Zertifizierung des sequentiellen Algorithmus auf und passen sie dann entsprechend auf einen verteilten Algorithmus hin an. Wir haben diese Fallstudie in [VR15] veröffentlicht.

Wir spezifizieren das Problem der Berechnung der kürzesten Pfade (Abschnitt 9.2.1) und geben eine zertifizierende verteilte Variante an, indem wir ein verteilbares Zeugenprädikat (Abschnitt 9.2.2), die Berechnung der Zeugen (Abschnitt 9.2.3) und einen Checker (Abschnitt 9.2.4) beschreiben.

9.2.1 Spezifikation des Problems

Wir gehen von einem Netzwerk $N = (V, E)$ mit einer Kostenfunktion über den Kanälen aus: $c : E \to \mathbb{R}_{>0}$. Für eine Komponente $s \in V$, genannt *Quelle*, sollen kürzeste Pfade zu allen Komponenten berechnet werden. Ein kürzester Pfad von einer Komponente zu einer anderen ist ein Pfad mit minimalen Kosten. Die Kosten eines Pfades $p = v_0, v_1, \ldots, v_n$ sind die Summe der Kosten der Kanäle des Pfades: $c(p) := \sum_{i=0}^{n-1} c((v_i, v_{i+1}))$

Gesucht ist also für jede Komponente $v \in V$:

$$d(v) := \min\{c(p) \mid p \text{ ist ein Pfad von } s \text{ nach } v\}$$

Die Pfade bilden dabei einen *Kürzeste-Pfade-Baum* mit der Quelle s als Wurzel.

9.2.2 Verteilbares Zeugenprädikat

Wir übernehmen das Zeugenprädikat des zertifizierenden sequentiellen Algorithmus zu Berechnung der kürzesten Pfade in einem Graphen [McC+11]. Der zertifizierende Algorithmus berechnet die

Funktion $D : V \to \mathbb{R}_{\geqslant 0}$ und bezeugt, dass die berechnete Funktion D der gesuchten Distanzfunktion d entspricht. Dafür beobachten wir, dass die berechnete Funktion D der gesuchten Distanzfunktion entspricht, falls in einem Netzwerk mit $s \in V$ als Quelle die folgenden Eigenschaften gelten:

$$D(s) = 0 \tag{9.1}$$

$$\text{für jedes } (u,v) \in E \text{ gilt } D(v) \leqslant D(u) + c(u,v) \tag{9.2}$$

$$\text{für jedes } v \neq s \text{ gibt es } (u,v) \in E \text{ mit } D(v) = D(u) + c(u,v) \tag{9.3}$$

9.2.2.1 *Zeugeneigenschaft*

Mit einem Induktionsbeweis über die Länge eines Pfades kann die Gleichheit $D = d$ gezeigt werden. Wir erläutern die Intuition der genannten Eigenschaften. Der leere Pfad hat die Kosten 0, es muss deswegen die Eigenschaft (9.1) gelten. Die Eigenschaft (9.2) besagt, dass es keinen Nachbarn u für eine Komponente v gibt über die ein kürzerer Pfad zur Quelle führt. Allerdings würde auch eine fehlerhafte Funktion D, die jeder Komponente eine zu kurze Distanz zur Quelle zuweist, die Eigenschaft (9.2) erfüllen. Entsprechend ist die Eigenschaft (9.3) zusätzlich notwendig, um die berechnete Distanz für jede Komponente zu rechtfertigen. Die berechnete Distanz D entspricht dann der gesuchten Distanz d und somit gilt die Zeugeneigenschaft.

Wir haben für die Zertifizierung der verteilten Berechnung der kürzesten Pfade das Zeugenprädikat des zertifizierenden sequentiellen Graphalgorithmus übernommen. Die Verteilungseigenschaft ist in diesem Fall jedoch bereits vorhanden.

9.2.2.2 *Verteilungseigenschaft*

Für jede der Eigenschaften (9.1) – (9.3) können wir jeweils ein lokales Prädikat angeben, sodass das Zeugenprädikat universell-verteilbar ist. Für die Eigenschaft (9.1) ist das lokale Prädikat für eine Komponente erfüllt, wenn die Komponente nicht die Quelle ist oder die berechnete Distanz 0 ist. Für die Eigenschaften (9.2) und (9.3) ist das lokale Prädikat jeweils erfüllt, wenn die Eigenschaft für Nachbarn gilt.

9.2.3 Berechnung der Zeugen

Die Berechnung der kürzesten Pfade gelingt mit der folgenden verteilten Variante des Bellman-Ford-Algorithmus aus [Lyn96].

9.2.3.1 *Verteilter Bellman-Ford-Algorithmus*

Jeder Knoten berechnet dabei seinen kürzesten Pfad zur Quelle s und stellt seinen Elternknoten fest. Initial gilt dabei $D(s) = 0$ für die Quelle und $D(v) = \infty$ für alle anderen Komponenten. Wenn eine Komponente ihre Distanz aktualisiert, sendet sie ihre aktuelle Distanz an all ihre Nachbarn. Jede Komponente v berechnet für jeden eingegangenen Wert $D(u)$ eines Nachbarn u das Minimum aus $D(v)$ und $D(u) + c(u, v)$ als aktuelle Distanz. Falls eine Komponente dabei ihre Distanz aktualisiert, wird der entsprechende Nachbar u neuer Elternknoten.

9.2.3.2 *Teilzeugen*

Für ihren Teilzeugen merkt sich jede Komponente bei der Berechnung den Nachbarn, mit dem die Eigenschaft (9.3) erfüllt ist. Im Gegensatz zu Zeugen des zertifizierenden sequentiellen Graphalgorithmus müssen Zeugen der verteilten Variante mehr beinhalten: Der Teilzeuge einer Komponente beinhaltet auch die berechnete Distanz eines jeden Nachbarn für die Prüfung der Eigenschaft (9.2).

9.2.4 Checker

Der Teilchecker der Quelle prüft die Eigenschaften (9.1) für seine Komponente. Er prüft also, dass die berechnete Distanz seiner Komponenten 0 ist. Ob es sich bei der Komponente um die Quelle handelt oder nicht, weiß ein Teilchecker durch die Teileingabe seiner Komponente.

Der Teilchecker einer Komponente, die nicht die Quelle ist, prüft die Eigenschaften (9.2) und (9.3) für seine Komponente. Die berechnete Distanz seiner Komponente kennt der Teilchecker durch deren Teilausgabe, während er die berechneten Distanzen der Nachbarn seiner Komponente aus dem Zeugen kennt. Die Kosten der Kanäle der Nachbarschaft wiederum sind ihm durch die Teileingabe seiner Komponente gegeben.

9.3 SPANNBÄUME

Die verteilte Konstruktion eines Spannbaums in einem Netzwerk ist ein Grundbaustein verteilter Algorithmen [AW04; Tel94; Ray13; Peloo; Lyn96; Gho14]. Eine Zertifizierung ist für uns entsprechend interessant.

Wir geben eine Spezifikation der Spannbaumkonstruktion an (Abschnitt 9.3.1). Anschließend geben wir eine zertifizierende Variante für die Spezifikation an, indem wir ein verteilbares Zeugenprädikat (Abschnitt 9.3.2), eine Berechnung der Zeugen (Abschnitt 9.3.3) und einen verteilten Checker (Abschnitt 9.3.4) beschreiben.

9.3.1 Spezifikation des Problems

Wir benutzen die folgende Charakterisierung für einen Spannbaum. Sei $N = (V, E)$ ein Netzwerk. Der Teilgraph $U = (V, E')$ mit $E' \subseteq E$ ist ein Spannbaum mit $r \in v$ als Wurzel, falls

Aufspannend: für jeden Knoten $v \in V$ gibt es einen r-v-Pfad mit Kanten aus E' und

Kreisfrei: $|E'| = |V| - 1$ gilt.

9.3.2 Verteilbares Zeugenprädikat

Wir geben zwei universell-verteilbare Prädikate an, Γ_1 und Γ_2, deren Konjunktion impliziert, dass der berechnete Spannbaum *aufspannend* ist und *kreisfrei* ist. Zusammen ergeben die beiden Prädikate dann entsprechend ein verteilbares Zeugenprädikat für die Spezifikation.

Für das erste Prädikat Γ_1 benutzen wir die Distanzfunktion, wie wir sie in Abschnitt 9.2 kennengelernt haben. Der Unterschied ist jedoch, dass wir die Distanz nicht für die kürzesten Pfade im Netzwerk benutzen, sondern für den berechneten Spannbaum. Die Kostenfunktion $c : E \to \{1\}$ ordnet dabei jedem Kanal die Kosten 1 zu. Die Wurzel r des Spannbaums ist dabei die Quelle.

Für die berechnete Distanz $D : V \to \mathbb{R}_{\geqslant 0}$ müssen nun die folgenden Eigenschaften gelten:

$$D(r) = 0 \tag{9.4}$$

$$\text{für } v \neq r \text{ gibt es einen Kanal } (u, v) \text{ mit } D(v) = D(u) + 1 \tag{9.5}$$

Das Prädikat Γ_1 ist erfüllt, falls die Eigenschaften (9.4) und (9.5) im Netzwerk gelten. Die Intuition der Eigenschaft (9.5) ist, dass der Nachbar u der Elternknoten im Spannbaum ist und somit (u, v) ein Kanal im Spannbaum ist, also $(u, v) \in E'$ aus der Spezifikation.

Das zweite Prädikat Γ_2 ist erfüllt, falls sich alle Komponenten über die Identität der Wurzel einig sind.

9.3.2.1 *Zeugeneigenschaft*

Für die Eigenschaft, dass der Baum *aufspannend* ist, argumentieren wir die Zeugeneigenschaft, wie folgt. Aus dem Prädikat Γ_1 folgt, wegen der Eigenschaft (9.5): Jede Komponente (bis auf die Wurzel) hat einen Nachbarn, der eine um 1 geringere Distanz zur Wurzel r hat. Es gibt also einen Pfad, auf dem die Distanz mit jeder Komponente um 1 kleiner wird. Wegen der Eigenschaft (9.4), folgt außerdem, dass der Pfad bei r endet.

Nun kann es aber weiterhin einen Wald, also mehrere Bäume mit unterschiedlichen Wurzeln im Netzwerk geben. Aus dem Prädikat Γ_2, der Einigkeit über die Identität der Wurzel, folgt dann jedoch, dass es nur eine Wurzel gibt. Der berechnete Spannbaum ist also aufspannend.

Betrachten wir nun die Eigenschaft, dass der berechnete Spannbaum kreisfrei ist. Aus dem Prädikat Γ_1 folgt, dass der berechnete Spannbaum kreisfrei ist. Schließlich kann die Distanz nicht immer kleiner werden, da Kanäle nur positive Kosten haben. Daraus folgt dann $|E'| \leqslant |V| - 1$.

Aus dem Prädikat Γ_2, der Einigkeit über die Wurzel r, folgt dann, dass es von jeder Komponente einen Pfad zur Wurzel gibt. Daraus folgt: $|E'| \geqslant |V| - 1$. Die Eigenschaft, dass der berechnete Spannbaum *kreisfrei* ist, folgt also aus den Prädikaten Γ_1 und Γ_2.

9.3.2.2 *Verteilungseigenschaft*

Das Prädikat Γ_1 ist analog zum Zeugenprädikat aus Abschnitt 9.2 universell-verteilbar.

Das Prädikat Γ_2 ist universell-verteilbar mit einem lokalen Prädikat, das für eine Komponente erfüllt ist, wenn es sich mit seinen Nachbarn über die Identität der Wurzel einig ist. Die Verteilungseigenschaft beruht darauf, dass wenn in jeder Nachbarschaft Einigkeit herrscht, dann auch im gesamten Netzwerk.

Beobachtung (ID-basiertes Netzwerk)**.** *Wir beobachten, dass es für die Zertifizierung eines Spannbaums in einem Netzwerk wichtig ist, von einem ID-basierten Netzwerk auszugehen. Andernfalls könnten wir die Eindeutigkeit der Wurzel nicht garantieren und hätten dann gegebenenfalls keinen Spannbaum, sondern einen aufspannenden Wald im Netzwerk.*

Wir weisen außerdem daraufhin, dass für anonyme Netzwerke auch eine von der Wurzel gewählte Zufallszahl, das Problem der Eindeutigkeit nicht löst. Zwei Wurzeln können schließlich die gleiche Zufallszahl wählen. In einem anonymen Netzwerk können wir einen Spannbaum also nicht zertifizieren. Die Variante mit der Zufallszahl könnte jedoch eine probabilistische Zertifizierung sein.

Beobachtung (Keine beschränkten Zeugen)**.** *Wir beobachten, dass die Identität der Wurzel immer globales Wissen sein muss. Entsprechend sind beschränkte Zeugen für die Zertifizierung eines Spannbaums nicht möglich.*

9.3.3 Berechnung des Zeugen

Der Teilzeuge einer Komponente enthält die eigene Distanz zur Wurzel, die Distanz seines Elternknotens, sowie die Identität der selbst gewählten Wurzel und die gewählten Wurzeln aller Nachbarn. Was also bei der zertifizierenden Berechnung eines Spannbaums mit berechnet werden muss, sind die Distanzen im Spannbaum. Dafür berechnet jede Komponente ihre Distanz so, dass sie um eins inkrementiert ist von der Distanz des gewählten Elternknotens.

9.3.4 Checker

Wie in Abschnitt 9.2, prüft der Teilchecker einer Komponente die Eigenschaft (9.4) im Fall der Wurzel und die Eigenschaft (9.5) ansonsten. Zur Entscheidung des Prädikats Γ_2, prüft der Teilchecker einer Komponente die Einigkeit über die Wurzel in der Nachbarschaft seiner Komponente.

9.4 BREITENSUCHE

Die Breitensuche auf einem Netzwerk ist ein Grundbaustein verteilter Algorithmen. Eine Zertifizierung der Breitensuche ist deswegen interessant. In [Lyn96, S.60–61] erläutert Lynch außerdem, wie Komponenten gemeinsam mithilfe einer Breitensuche im Netzwerk eine globale Funktion berechnen.

Eine Breitensuche auf einem zusammenhängenden Graphen erzeugt einen Breitensuchebaum. Wir beobachten, dass für eine Kostenfunktion, die jeder Kante die gleiche positive Zahl zuordnet, ein Breitensuchebaum ein Kürzester-Pfade-Baum ist. Infolgedessen können wir die zertifizierende Variante zur verteilten Konstruktion eines Kürzeste-Pfade-Baums (siehe Abschnitt 9.2) für die verteilte Breitensuche übernehmen.

9.5 KONSENSFINDUNG

Die Konsensfindung ist ein häufiges Problem in verteilten Systemen. Es gibt entsprechend einige Varianten des Problems, sowie verteilte

Algorithmen, die das Problem der Konsensfindung lösen [AW04; Tel94; Ray13; Pel00; Lyn96; Gho14].

Wir geben eine Spezifikation der Konsensfindung an (Abschnitt 9.5.1). Anschließend geben wir eine zertifizierende Variante der Konsensfindung für diese Spezifikation an, indem wir ein verteilbares Zeugenprädikat (Abschnitt 9.5.2), eine Berechnung der Zeugen (Abschnitt 9.5.3) und einen verteilten Checker (Abschnitt 9.5.4) beschreiben.

9.5.1 Spezifikation

Bei der Konsensfindung geht es um das Problem, dass sich alle Komponenten eines Netzwerks auf eine Wahl aus gegebenen Optionen einigen. Obwohl das Problem vielleicht einfach erscheinen mag, gibt es ausgeklügelte verteilte Algorithmen für die Konsensfindung. Die Schwere des Problems liegt in der Verteiltheit der Wahl. Was die Optionen sind, die zur Auswahl stehen, ist für die allgemeine Betrachtung des Problems unwichtig.

Eine Konsensfindung in einem Netzwerk ist korrekt, falls die beiden folgenden Eigenschaften gelten:

Einigkeit: Jede Komponente des Netzwerks trifft die gleiche Wahl und

Validität: die getroffene Wahl ist eine Option, die zur Auswahl steht.

Darüber hinaus gibt es manchmal noch Anforderungen dazu, *wie* ein Konsens gefunden wird. Zum Beispiel, dass die getroffene Wahl von mindestens einer Komponente als Option auch vorgeschlagen wurde.

Mit zertifizierenden Algorithmen verifizieren wir jedoch funktionale Korrektheit, also das *Was* und nicht das *Wie*. Die Konstruktion betrachten wir als Black-Box. Für die Konsensfindung bedeutet das, dass wir uns dafür interessieren, dass es einen Konsens gibt und nicht wie er zu Stande gekommen ist.

Dennoch können wir die oben genannte Eigenschaft mit einem Spannbaum bezeugen. Die Wurzel des Spannbaums ist die Komponente, die die getroffene Wahl als Option vorgeschlagen hat. Die verteilte Zertifizierung eines Spannbaums haben wir im vorigen Abschnitt 9.3 bereits gesehen.

9.5.2 Verteilbares Zeugenprädikat

Wir geben für die Eigenschaften der *Einigkeit* und der *Validität* jeweils ein universell-verteilbares Prädikat an, dass die jeweilige Eigenschaft

impliziert. Die Konjunktion dieser Prädikat ist das verteilbare Zeugenprädikat.

Wir betrachten zunächst die Eigenschaft der Einigkeit aller Komponenten eines Netzwerks. Die Einigkeit wird durch ein universell-verteilbares Prädikat bezeugt; das lokale Prädikat ist für eine Komponente erfüllt, falls die Komponente die gleiche Wahl, wie ihre 1-Nachbarn getroffen hat. Die Verteilungseigenschaft beruht darauf, dass wenn sich Nachbarn jeweils einig sind, dass dann auch Einigkeit im Netzwerk gilt.

Die Validität gilt, wenn für jede Komponente gilt, dass ihre getroffene Wahl als Option zur Auswahl steht. Das Prädikat ist also ein universell-verteilbares Zeugenprädikat mit offensichtlicher Verteilungs- und Zeugeneigenschaft.

Die Zeugeneigenschaft für die Konjunktion der Prädikate zur Einigkeit und Validität folgt unmittelbar.

9.5.3 Berechnung der Zeugen

Der Teilzeuge einer Komponente ist die eigene Wahl, sowie die Wahl eines jeden Nachbarn. Dieser Teilzeuge ist für die Einigkeit nötig, für die Validität ist kein Teilzeuge nötig, aber der Checker muss die gesamte Auswahl der Optionen kennen.

Bei verteilten Algorithmen zur Konsensfindung schicken die Komponenten ihre Wahl auch ihren Nachbarn. Die Berechnung der Zeugen ist deswegen einfach integrierbar.

9.5.4 Checker

Für die Überprüfung der Einigkeit vergleicht der Teilchecker die Wahl seiner Komponente mit der Wahl der Nachbarn. Für die Validität muss jeder Teilchecker die gesamte Auswahl kennen.

9.6 BROADCAST

Bei einem Broadcast handelt es sich um einen Grundbaustein verteilter Algorithmen [AW04; Tel94; Ray13; Pel00; Lyn96; Gho14]. Eine Zertifizierung ist somit interessant.

Wir spezifizieren das Problem (Abschnitt 9.6.1) und geben eine zertifizierende verteilte Variante an, indem wir ein verteilbares Zeugenprädikat (Abschnitt 9.6.2), die Berechnung der Zeugen (Abschnitt 9.6.3) und einen Checker (Abschnitt 9.6.4) beschreiben.

9.6.1 Spezifikation des Problems

Bei dem Broadcast-Problem soll eine Nachricht von einer Komponente des Netzwerks ausgehend an alle anderen Komponenten kommuniziert werden. Wir nennen die Komponente, von der die Nachricht ausgeht, den *Initiator*. Das Broadcast-Problem ist korrekt gelöst, falls alle Komponenten die Nachricht des Initiators erhalten haben.

9.6.1.1 *Weitere Anforderungen*

Darüber hinaus wird bei einem Broadcast manchmal noch gefordert,

- dass die Nachricht nicht unnötig im Kreis geschickt wird und
- dass der Initiator informiert wird, dass alle Komponenten seine Nachricht erhalten haben.

9.6.2 Verteilbares Zeugenprädikat

Wir stellen fest, dass gilt: wenn sich alle Komponenten, inklusive des Initiators, über den Inhalt der empfangenen Nachricht *einig* sind, dann haben auch alle Komponenten die Nachricht des Initiators erhalten.

Bei der zertifizierenden Konstruktion eines Spannbaums (siehe Abschnitt 9.3) haben wir ein universell-verteilbares Prädikat für die Einigkeit über die Wurzel des Spannbaums gesehen. Der einzige Unterschied hier ist, dass sich die Komponenten über eine Nachricht einig sind.

Im Falle einer großen Nachricht ist der Abgleich zwischen Nachbarn über den Inhalt der Nachricht aufwändig. Eine praktische Lösung könnte hier eine Nummerierung der Nachricht durch den Initiator sein. Statt des Inhalts wird somit nur die Nummer der Nachricht in Nachbarschaften abgeglichen. Jedoch ist die Korrektheit des Broadcast nur garantiert, wenn die Nummerierung von Nachrichten eindeutig ist.

9.6.2.1 *Reduktion auf Spannbaumkonstruktion*

Für die weiteren Anforderung an einen Broadcast betrachten wir bei der Berechnung der Zeugen (Abschnitt 9.6.3) den Echo-Algorithmus, wie er in [Lyn96] beschrieben ist. Der Echo-Algorithmus löst das Broadcast-Problem in einem asynchronen Netzwerk, indem er einen Spannbaum mit dem Initiator als Wurzel aufbaut. Der Spannbaum ist dann die Kommunikationsstruktur für die Nachricht des Initiators. Dadurch ist sichergestellt, dass die Nachricht nicht unnötig im Kreis geschickt wird und die Komponenten jeweils an ihren Elternknoten den Erhalt der Nachricht quittieren können.

Wir reduzieren die weiteren Anforderungen an einen Broadcast also auf die Konstruktion eines Spannbaums und benutzen dafür entsprechend das verteilbare Zeugenprädikat aus Abschnitt 9.3.

9.6.3 Berechnung der Zeugen

Für die Eigenschaft, dass jede Komponente die Nachricht des Initiators erhalten hat, benötigt eine Komponente die Nachricht der Nachbarn als Teilzeugen.

Für die weiteren Anforderungen ist der Zeuge ein Spannbaum im Netzwerk. Der Echo-Algorithmus löst das Broadcast-Problem und berechnet dabei einen Spannbaum.

9.6.3.1 *Echo-Algorithmus*

Der Initiator sendet seine Nachricht an all seine Nachbarn. Jede Komponente, die eine Nachricht von mindestens einem Nachbarn erhält, wählt einen dieser Nachbarn als Elternknoten. Anschließend sendet die Komponente die Nachricht an all ihre Nachbarn mit Ausnahme ihres Elternknotens. Eine Komponente sendet ihrem Elternknoten eine Bestätigung, falls sie von allen Nachbarn eine Nachricht erhalten. Bei den Nachrichten der Nachbarn darf es sich dabei jeweils entweder um die ursprüngliche Nachricht oder eine Bestätigung handeln. Blätter erhalten nur die ursprüngliche Nachricht.

Der Echo-Algorithmus terminiert damit, dass der Initiator am Ende von all seinen Nachbarn eine Nachricht erhält. Von den Nachbarn, die ihn als Elternknoten gewählt haben, erhält er eine Bestätigung und von den restlichen Nachbarn erhält er die Nachricht, die er selbst verschickt hat. Er erfährt dadurch, dass alle Komponenten die Nachricht erhalten haben.

Der Echo-Algorithmus konstruiert also im Netzwerk einen Spannbaum mit dem Initiator als Wurzel. Abgesehen von der Wurzel sieht der Spannbaum beliebig aus, weil jede Komponente ihren Elternknoten unter ihren Nachbarn beliebig wählt.

Die Teilzeugen sind für die Zertifizierung des Spannbaums, wie in Abschnitt 9.3 beschrieben, aufgebaut.

9.6.4 Checker

Die Teilchecker der Komponenten sehen, wie in Abschnitt 9.3 beschrieben, aus.

9.7 LEADER ELECTION

Leader-Election ist ein Grundbaustein verteilter Algorithmen [AW04; Tel94; Ray13; Pel00; Lyn96; Gho14]. Eine Zertifizierung ist somit interessant. Wir haben diese Fallstudie unter anderem in [VA17] veröffentlicht.

Wir spezifizieren das Problem (Abschnitt 9.7.1) und geben eine zertifizierende verteilte Variante an, indem wir das Problem der Leader-Election auf das Problem der Spannbaumkonstruktion reduzieren (Abschnitt 9.7.2).

9.7.1 Spezifikation des Problems

Wir haben das Problem der Leader-Election bereits in Abschnitt 5.5 spezifiziert. Das Problem der Leader-Election ist korrekt gelöst, falls der Leader eindeutig gewählt ist und der Leader eine Komponente des Netzwerks ist.

9.7.2 Reduktion auf Spannbaumkonstruktion

Ein Spannbaum mit dem gewählten Leader als Wurzel bezeugt, dass der Leader eindeutig gewählt ist, da sich alle Komponenten einig sind über die Wurzel des Spannbaums. Der Spannbaum bezeugt außerdem die Existenz des Leaders im Netzwerk und damit, dass der Leader eine Komponente des Netzwerks ist.

Verteilte Algorithmen zur Leader-Election, die ohnehin einen solchen Spannbaum berechnen, gibt es einige [AW04; Tel94; Ray13; Lyn96].

9.8 BIPARTITHEITSTEST

Ein Färbbarkeitstest mit einer bestimmten Anzahl an Farben für die Komponenten in einem Netzwerk ist ein Grundbaustein verteilter Algorithmen [AW04; Tel94; Ray13; Pel00; Lyn96; Gho14; Erc13]. Dabei haben Nachbarn nie die gleiche Farbe und deswegen ist die Färbbarkeit zum Beispiel interessant, um eine Symmetrie zwischen Nachbarn zu brechen, um beim gemeinsamen Rechnen Ressourcen oder Aufgaben zu verteilen oder auch damit Nachbarn in verschiedenen Frequenzen senden und somit zur Reduktion der Interferenz in kabellosen Netzwerken beitragen [BK18; Abd+14; HA11].

In [McC+11] gibt es bereits einen zertifizierenden sequentiellen Bipartitheitstest für Graphen, also einen Färbbarkeitstest mit zwei Farben.

Eine Zertifizierung eines verteilten Bipartitheitstest ist deswegen für uns aus zwei Gründen interessant. Zum einen handelt es sich um einen grundlegenden verteilten Algorithmus und zum anderen gibt es bereits eine zertifizierende sequentielle Variante.

Wir haben die Fallstudie unter anderem in [Völ17] veröffentlicht.

Wir spezifizieren das Problem (Abschnitt 9.8.1) und geben eine zertifizierende Variante an, indem wir ein verteilbares Zeugenprädikat (Abschnitt 9.8.2), die Berechnung der Zeugen (Abschnitt 9.8.3) und einen Checker (Abschnitt 9.8.4) beschreiben.

9.8.1 Spezifikation des Problems

In Abschnitt 6.1 haben wir für einen verteilten Bipartitheitstest bereits die Zertifizierung für den Fall eines bipartiten Netzwerks als einführendes Beispiel beschrieben. Wir betrachten deswegen im folgenden nur den Fall eines nicht bipartiten Netzwerks.

Der Bipartitheitstest ist für nicht bipartites Netzwerk N korrekt gelöst, falls

- die verteilte Ausgabe ein „nein“ enthält und
- N nicht bipartit ist.

9.8.2 Verteilbares Zeugenprädikat

Das verteilbare Zeugenprädikat ist eine Disjunktion der verteilbaren Prädikate für den Fall eines bipartiten und den Fall eines nichtbipartiten Netzwerks.

Wir betrachten nun den Fall eines nicht bipartiten Netzwerks. In der zertifizierenden sequentiellen Variante aus [McC+11] bezeugt ein ungerader Kreis, dass ein Graph nicht bipartit ist. Das heißt, das Zeugenprädikat ist erfüllt, wenn die Ausgabe „nein“ ist und ein ungerader Kreis als Teilgraph im Eingabegraphen enthalten ist. Wir übernehmen die Idee dieses Zeugenprädikats für Netzwerke.

9.8.2.1 *Zeugeneigenschaft*

Ein ungerader Kreis ist selbst nicht bipartit. Denn die Knoten eines Kreises $v_0, v_1, ..., v_{2k+1}$ mit $v_0 = v_{2k+1}$ müssten dafür abwechselnd gefärbt werden. Der Knoten v_0 müsste dann beide Farben annehmen. Folglich ist ein ungerader Kreis nicht bipartit. Weiterhin gilt, dass wenn ein nicht bipartiter Teilgraph in einem Graphen enthalten ist, dass dann der ganze Graph nicht bipartit ist.

Für ein Netzwerk gilt also, wenn es einen ungeraden Kreis enthält, dann ist das Netzwerk nicht bipartit. Wir betrachten den zertifizierenden verteilten Bipartitheitstest hier aus der Perspektive eines Nutzers; wir beweisen deswegen auch nicht die Vollständigkeit des Zeugenprädikats. Wir weisen dennoch daraufhin, dass sich der Beweis der Vollständigkeit – es gibt immer einen ungeraden Kreis in einem nicht bipartiten Netzwerk – aus dem Theorem von König und Ergervary ergibt [Azi11].

9.8.2.2 *Verteilungseigenschaft*

Wir gestalten das Zeugenprädikat Γ mithilfe dreier Verteilungsprädikate verteilbar. Es gilt dann, das die Konjunktion der Verteilungsprädikate das Zeugenprädikat Γ impliziert. Wir machen zunächst einige Beobachtungen, um die Prädikate Γ_1, Γ_2 und Γ_3 herzuleiten.

Nehmen wir an, wir haben einen Spannbaum in einem Netzwerk gegeben. Ein Baum ist immer bipartit, da wir die Ebenen abwechselnd färben können. Wir nehmen deswegen weiterhin an, dass neben dem Spannbaum auch eine Bipartition des Spannbaums gegeben ist.

Wir nutzen den Spannbaum und dessen Bipartition, um ein verteilbares Zeugenprädikat für die Existenz eines ungeraden Kreises im Netzwerk zu formulieren:

Theorem 9.8.1. *Gegeben sei ein durch eine Bipartition zwei-gefärbter Spannbaum in einem Netzwerk. Wenn mindestens eine Komponente einen Nachbarn gleicher Farbe hat, dann gibt es einen ungeraden Kreis im Netzwerk.*

Beweis. Sei $N = (V, E)$ ein Netzwerk und $T = (V, E')$ ein Spannbaum in N mit einer Bipartition (V_1, V_2) der Komponenten V für E'. Nehmen wir nun an es gibt eine Komponente u mit einem Nachbarn v gleicher Farbe. Ohne Beschränkung der Allgemeinheit nehmen wir an $u, v \in V_1$. Für den Kanal (u, v) gilt dann $(u, v) \notin E'$, denn (V_1, V_2) ist für E' eine Bipartition von V.

Da u und v Teil des Spannbaums sind, gibt es einen Pfad mit den Kanälen des Spannbaums, der u und v verbindet. Gemeinsam ergeben der Pfad über die Kanäle des Spannbaums und der Kanal (u, v) einen Kreis.

Für die Komponenten u und v gilt, dass sie entweder einen gemeinsamen Vorfahren k im Spannbaum haben (1) oder dass eine der Komponenten Vorfahre der anderen ist (2). Die Abbildung 9.1 stellt die beiden Fälle grafisch dar.

Im Fall (1) gilt, dass u und v entweder beide geraden Abstand vom gemeinsamen Vorfahren k haben oder aber ungeraden, da sie die gleiche Farbe haben. Der Pfad zwischen u und v über Kanäle des

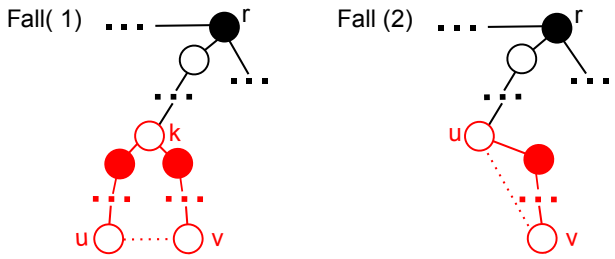


Abbildung 9.1: r ist die Wurzel des Spannbaums. u und v sind durch einen Kanal benachbart, der nicht zum Spannbaum gehört (gestrichelte rote Linie). Der ungerade Kreis ist in rot hervorgehoben.

Spannbaums hat also gerade Länge und bildet gemeinsam mit dem Kanal (u, v) einen ungeraden Kreis.

Für Fall (2) nehmen wir ohne Beschränkung der Allgemeinheit an, dass u Vorfahre von v ist. Da u und v die gleiche Farbe haben, hat der Pfad über die Kanäle des Spannbaum gerade Länge und bildet gemeinsam mit dem Kanal (u, v) einen ungeraden Kreis. □

Wir können nun die drei Verteilungsprädikate Γ_1, Γ_2 und Γ_3 angeben, die gemeinsam das Zeugenprädikat – es gibt einen ungeraden Kreis im Netzwerk – implizieren. Γ_1 ist erfüllt, falls das Netzwerk einen Spannbaum S enthält. Γ_2 ist erfüllt, falls eine Bipartition für S vorliegt. Γ_3 ist erfüllt, wenn mindestens eine Komponente einen Nachbarn mit anderer Farbe hat.

Das Verteilungsprädikate Γ_1 ist, wie wir in Abschnitt 9.3 gesehen haben universell-verteilbar mithilfe einer Charakterisierung über die Distanzfunktion. Das Verteilungsprädikat Γ_2 ist auch universell-verteilbar. Wir haben es bereits in unserem einführenden Beispiel in Abschnitt 6.1 betrachtet.

Das Verteilungsprädikat Γ_3 ist existenziell-verteilbar mit einem lokalen Prädikat, dass für eine Komponente erfüllt ist, wenn sie einen Nachbarn anderer Farbe hat.

9.8.2.3 *Vollständige Verteilbarkeit*

Wir betrachten den zertifizierenden verteilten Bipartitheitstest hier aus der Perspektive eines Nutzers; wir beweisen deswegen auch nicht die vollständige Verteilbarkeit des Zeugenprädikats. Wir skizzieren dennoch kurz, was dafür zu tun wäre.

Für die Vollständigkeit muss gezeigt folgendes werden. Wenn es einen ungeraden gibt, dann gibt es (unter Voraussetzung eines bipartiten Spannbaums) immer eine Komponente mit Nachbarn anderer Farbe.

Dass es immer einen Spannbaum mit Bipartition gibt, gilt unabhängig davon, ob es ein bipartites oder nicht bipartites Netzwerk ist.

Das Theorem, dass wir für die Vollständigkeit zeigen müssen, ist also folgendes:

Theorem 9.8.2. *Sei ein durch eine Bipartition zwei-gefärbter Spannbaum in einem Netzwerk gegeben. Es gilt:*

(i) Jeder ungerade Kreis in dem Netzwerk enthält mindestens eine Kante, die nicht zum Spannbaum gehört.

(ii) Wenn es einen ungeraden Kreis im Netzwerk gibt, der mehr als eine Kante enthält, dann existiert auch ein ungerader Kreis mit nur einer Kante, die nicht zum Spannbaum gehört.

9.8.3 Berechnung der Zeugen

Einen verteilten Bipartitheitstest finden wir zum Beispiel in [CH+16]. Es wird dabei ohnehin ein Spannbaum berechnet. Die Bipartition wählen wir dann, indem wir die Distanz beim Spannbaum benutzen. Alle Komponenten mit gerader Distanz haben die eine und alle mit ungerader Distanz zur Wurzel die andere Farbe. Damit haben wir alle Informationen, die wir für den Zeugen brauchen.

9.8.4 Checker

Die Teilchecker ergeben sich aus den Teilcheckern bei der Zertifizierung des Spannbaums (Abschnitt 9.3) und den Teilcheckern des einführenden Beispiels (Abschnitt 6.1).

Teil IV

VERTEILTE CHECKER

Wir widmen uns in diesem Teil der Arbeit verteilten Checkern. Bisher haben wir die Entscheidung der lokalen Prädikate eines verteilbaren Zeugenprädikats je Fallstudie betrachtet (siehe Kapitel 9).

In Kapitel 10 stellen wir die verteilte Prüfung verteilbarer Zeugenprädikate vor, sowie die dafür nötige Konsistenzprüfung verteilter Zeugen.

In Kapitel 11 stellen wir eine industrielle Fallstudie vor, in deren Rahmen wir einen verteilten Algorithmus für ein Multi-Agenten-System zertifizierend gestalten. Darüber hinaus reduzieren wir die Invasivität des Checkers durch den Einsatz virtueller Netzwerke.

10 VERTEILTE PRÜFUNG VERTEILBARER ZEUGENPRÄDIKATE

In diesem Kapitel betrachten wir die verteilte Prüfung verteilbarer Zeugenprädikate, sowie die dafür nötige Konsistenzprüfung verteilter Zeugen durch verteilte Checker.

In Abschnitt 10.1 diskutieren wir zunächst die gewählte verteilte Architektur der Checker.

In Abschnitt 10.2 stellen wir die Evaluation verteilbarer Zeugenprädikate in Netzwerken vor und diskutieren Eigenschaften der Evaluation in Abschnitt 10.3.

In Abschnitt 10.4 betrachten wir die Konsistenzprüfung verteilter Zeugen. Ergebnisse dieses Abschnitts haben wir in [VA18] veröffentlicht.

10.1 VERTEILTE ARCHITEKTUR

In diesem Abschnitt betrachten wir die gewählte verteilte Architektur der Checker, die wir bereits vorgreifend erwähnt haben (siehe Kapitel 6, Abschnitt 6.4).

In Abschnitt 10.1.1 stellen wir zunächst Anforderungen an verteilte Checker vor. In Abschnitt 10.1.2 gehen wir dafür auf die Architektur der Teilchecker ein. In Abschnitt 10.1.3 beschreiben wir die Initialisierung verteilter Checker.

10.1.1 Anforderungen an verteilte Checker

In Kapitel 8 (siehe Abschnitt 8.4) haben wir die Güte zertifizierender verteilter Algorithmen diskutiert. Dabei ist ein wichtiges Kriterium die Prüfbarkeit eines verteilbaren Zeugenprädikats. In diesem Zusammenhang haben wir schon in Kürze auch Anforderungen an verteilte Checker betrachtet. Wir stellen folgend in Kürze die spezifischen Anforderungen an verteilte Checker heraus.

10.1.1.1 *Einfache formale Verifikation*

Der Nutzer einer zertifizierenden Variante eines verteilten Algorithmus muss auf die Korrektheit des zugehörigen Checkers vertrauen. Es ist daher hilfreich, wenn der implementierte Checker sich mit möglichst geringem Aufwand formal verifizieren lässt.

10.1.1.2 *Verteiltheit, Gleichheit, Lokalität*

Verteiltheit, Gleichheit und Lokalität sind Kerngedanken verteilter Algorithmen, auf die wir auch bei der Entwicklung verteilter Checkern achten sollten.

Für die Lokalität erscheint darüber hinaus eine zusätzliche Betrachtung interessant. Eine Komponente kommuniziert während eines verteilten Algorithmus Informationen in einem gewissen Teil eines Netzwerks. Insbesondere in sicherheitskritischen Anwendungsfällen können wir uns vorstellen, dass eventuell nicht alle Informationen im gesamten Netzwerk geteilt werden sollten. In diesem Fall sollte der Teilchecker einer Komponente die Informationen auch nur in dem entsprechenden Teil des Netzwerks teilen.

10.1.1.3 *Geringe Invasivität*

Invasivität (übersetzt vom englischen intrusiveness) beschreibt den Grad zu welchem eine Laufzeitverifikation in das ursprüngliche System eingreift. Eine geringe Invasivität ist ein gängiges Kriterium für die Laufzeitverifikation verteilter Systeme [FPS18].

Checker sollten so wenig wie möglich in das System eingreifen, da jeder Eingriff neue Fehlerquellen in das System einführt. Wenn ein Checker zum Beispiel viele Nachrichten verschickt oder aufwändige Berechnungen durchführt, dann macht er ein verteiltes System damit anfälliger für Fehler, wie Nachrichtenverlust oder den Crash einer Komponente. Checker sollten deswegen möglichst wenige Ressourcen verbrauchen und damit möglichst wenig invasiv in das System einzugreifen.

Wir betrachten die Invasivität eines Checkers im Detail in der industriellen Fallstudie in Kapitel 11.

10.1.2 Architektur der Teilchecker

Die Abbildung 10.1 illustriert die gewählte verteilte Architektur für Checker; sie zeigt die Integration eines Teilcheckers am Beispiel einer Komponente. Wir sehen im oberen Teil der Abbildung ein Netzwerk N mit der Komponente v und ihren Nachbarn u_1, u_2 und u_3.

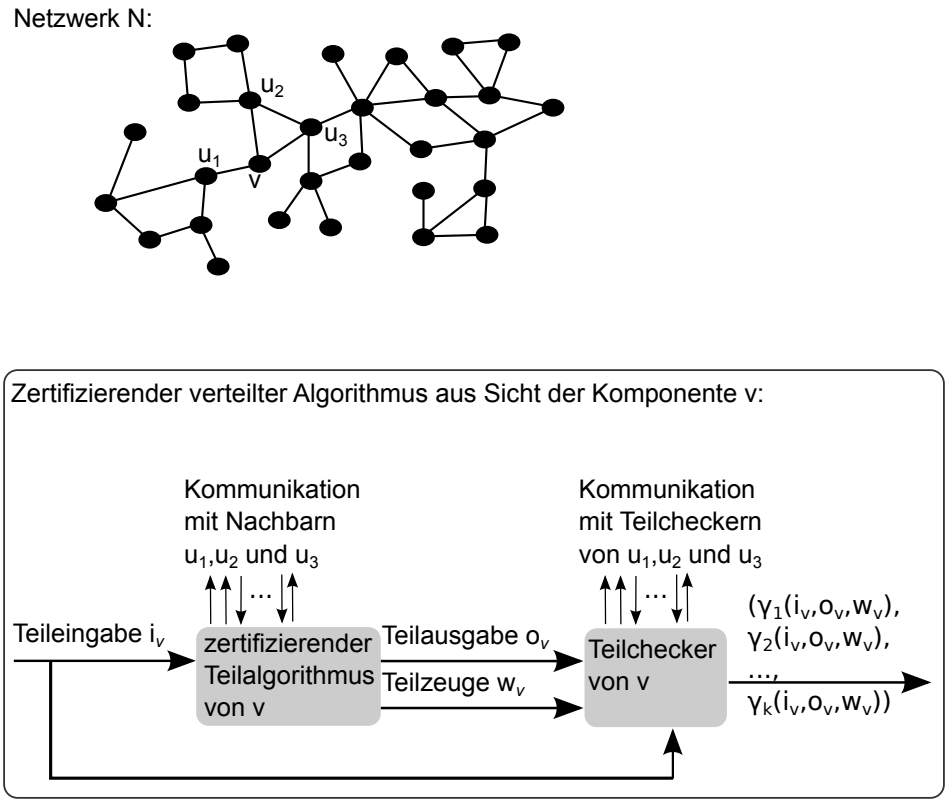


Abbildung 10.1: Netzwerk N mit der Komponente v und ihren Nachbarn u_1, u_2 und u_3. Integration des Teilcheckers am Beispiel der Komponenten v mit Teileingabe i_v, Teilausgabe o_v, Teilzeuge w_v und lokalen Prädikaten $\gamma_1, \gamma_2, ..., \gamma_k$. Die oberen ein- und ausgehenden Pfeile des zertifizierenden Teilalgorithmus und des Teilcheckers von v stellen eine beliebige Kommunikation mit Nachbarn dar.

10.1.2.1 *Teilchecker als logische Einheit*

Ein Teilchecker ist eine eigene *logische Einheit*, die einer Komponente des Netzwerks zugeordnet ist. Ein Teilchecker kann dabei auf der entsprechenden Netzwerkkomponente ausgeführt werden oder aber auch auf einer Komponente, die eigens dafür ins Netzwerk integriert wird. Auch für die Komponenten des Netzwerks haben wir nicht festgelegt, ob es sich bei einer Komponente um logische Einheit oder eine physikalische Einheit handelt. Diese abstrakte Betrachtung von Netzwerken ist üblich [AW04; Tel94; Ray13; Pel00; Lyn96; Erc13; Gho14].

10.1.2.2 *Entscheidung der lokalen Prädikate*

Die Kommunikation zwischen einem zertifizierenden Teilalgorithmus und dem zugehörigen Teilchecker verläuft analog zur Kommunikation eines zertifizierenden sequentiellen Algorithmus mit dessen Checker.

Der zertifizierende Teilalgorithmus einer Komponente kommuniziert Teilausgabe, sowie Teilzeugen an ihren Teilchecker. Der Teilchecker erhält außerdem auch die Teileingabe.

Hierbei stoßen wir auf ein Problem, das für zertifizierende sequentielle Algorithmen bereits bekannt ist [McC+11]. Die Teileingabe, die der Teilchecker erhält muss eine vertrauenswürdige Kopie sein. Um dies sicherzustellen werden mitunter kryptographische Methoden eingesetzt. In [ZPK14] wird dies in Kombination mit einem zertifizierenden sequentiellen Algorithmus genutzt.

Mit der Teileingabe, der Teilausgabe und dem Teilzeugen entscheidet der Teilchecker einer Komponente dann alle lokalen Prädikate $(\gamma_1, \gamma_2, ..., \gamma_k)$ des verteilbaren Zeugenprädikats. Die lokalen Prädikate und ihre Entscheidungsprozedur sind abhängig vom Problem. Wir haben sie deswegen auch in den Fallstudien aus Kapitel 9 einzeln betrachtet.

10.1.2.3 *Kommunikation der Teilchecker*

Eine Komponente kann ihren Nachbarn in einem Netzwerk direkt Nachrichten schicken. Wir integrieren Teilchecker so, dass auch ein Teilchecker einer Komponente mit den Teilchecker der benachbarten Komponenten direkt kommunizieren kann. Ein Austausch zwischen benachbarten Teilcheckern ist zum Beispiel für die Prüfung der Konsistenz eines Zeugen nötig (siehe Abschnitt 10.4).

Wir wählen eine direkte Kommunikation zwischen benachbarten Teilcheckern. Der Grund dafür ist, dass eine indirekte Kommunikation nicht vertrauenswürdig wäre, da die Komponenten selbst nicht vertrauenswürdig sind. Der Beweis dieser Aussage ergibt sich aus dem Beweis des Theorems 6.4.1 (siehe Abschnitt 6.4), indem wir uns mit der nötigen Zentralität sequentieller Checker befasst haben.

Der entscheidende Unterschied für die Vertrauenswürdigkeit einer direkten Kommunikation der Teilchecker ist die Vertrauenswürdigkeit der Teilchecker; für diese muss eine Entwicklerin entsprechend Sorge tragen. Wie das gelingt erläutern wir in Teil v zur formalen Instanzverifikation.

Eine alternative Integration ist den Teilchecker einer Komponente als ein Interface zu wählen, sodass er alle Nachrichten mitliest. Ein Nachteil dieser Alternative ist, dass durch das Mitlesen während der Konstruktion, die Konstruktion weniger stark von der Prüfung getrennt ist.

Wir halten die hier vorgeschlagene verteilte Architektur für praxistauglich. Da ein Teilchecker eine logische Einheit ist und einer Komponente des Netzwerks zugeordnet ist, sollte die Kommunikation benachbarter Teilchecker in der Praxis gut umsetzbar sein. Wir stellen in Kapitel 16

eine Simulationsumgebung für zertifizierende verteilte Algorithmen vor, die diese Kommunikation mithilfe üblicher Simulationswerkzeuge auch implementiert [Aki15].

10.1.3 Initialisierung

Für die Integration der Teilchecker ist es nötig, diese wie die Komponenten eines Netzwerks zu initialisieren.

10.1.3.1 *Wissen zur Topologie*

Die Eingabe eines verteilten Algorithmus ist unter anderem auch das Netzwerk selbst (siehe Kapitel 5). Das heißt jede Komponente erhält als Teileingabe ihre eigene ID, sowie die IDs der Nachbarn. Bei einem ID-basierten Netzwerk wird davon ausgegangen, dass dieses Wissen zur Topologie des Netzwerks durch eine Initialisierung allen Komponenten bekannt ist. Wir gehen deswegen davon aus, dass auch die Teilchecker der Komponenten initialisiert werden.

10.1.3.2 *Rolle der Vorbedingung*

Akzeptiert ein Checker ein Tripel bestehend aus Eingabe, Ausgabe und Zeuge, so kann das auch bedeuten, dass die Vorbedingung für die Eingabe verletzt ist. Bei einem „normal“ zertifizierenden Algorithmus gelten alle Korrektheitsaussagen unter der Annahme, dass die Eingabe die Vorbedingung erfüllt. Bei zertifizierenden sequentiellen Algorithmen existiert das Konzept eines stark zertifizierenden Algorithmen, der zusätzlich bezeugt, dass die Eingabe die Vorbedingung erfüllt.

Überträgt man das Konzept der starken Zertifizierung auf verteilte Algorithmen, so müsste ein stark zertifizierender verteilter Algorithmus unter anderem für die Eingabe bezeugen, dass es sich bei der Eingabe tatsächlich um das zugrunde liegende Netzwerk handelt.

In [Ray13] finden wir verteilte Algorithmen, mit denen die Komponenten eines Netzwerks die Topologie des Netzwerks lernen. Für eine Zertifizierung der Topologie müssen sich die benachbarten Teilchecker der Komponenten einig sein.

10.2 EVALUATION VERTEILBARER ZEUGENPRÄDIKATE

Die Entscheidung eines verteilbaren Zeugenprädikats findet durch die Summe der Entscheidungen der lokalen Prädikate statt. Wir sprechen

von der Evaluation eines verteilbaren Zeugenprädikats, da nur noch Wahrheitswerte logisch kombiniert werden.

Wir spezifizieren das Problem der Evaluation in Abschnitt 10.2.1 und geben einen verteilten Algorithmus zur Evaluation in Abschnitt 10.2.2 an.

10.2.1 Spezifikation des Problems der Evaluation

Wir unterscheiden zwei Fälle bei der Spezifikation des Problems der Evaluation eines verteilbaren Zeugenprädikats in einem Netzwerk. Die Evaluation ist korrekt in einem Netzwerk gelöst, wenn

Fall 1) zentraler Nutzer: eine (ausgewählte) Komponente den Wahrheitswert des verteilbaren Zeugenprädikats kennt.

Fall 2) verteilter Nutzer alle Komponenten den Wahrheitswert des verteilbaren Zeugenprädikats kennen.

Die beiden Fälle spiegeln jeweils die Laufzeitverifikation für einen zentralen Nutzer und für einen verteilten Nutzer wider (siehe Kapitel 5 auf Seite 50).

10.2.2 Verteilter Algorithmus zur Evaluation

Wir beschreiben einen verteilten Algorithmus zur Evaluation. Dafür nehmen wir zunächst einen gewurzelten Spannbaum im Netzwerk an. In Abschnitt 10.3.2 diskutieren wir diese Annahme.

Sei Γ ein verteilbares Zeugenprädikat mit Verteilungsprädikaten $\Gamma_1, \Gamma_2, .., \Gamma_k$. Für jedes Verteilungsprädikat Γ_j mit $j = 1, .., k$ gilt dann, dass es entweder universell-verteilbar oder existenziell-verteilbar ist mit einem lokalen Prädikat γ_j. Jeder Teilchecker weiß, wie das verteilbare Zeugenprädikat aufgebaut ist.

10.2.2.1 *Verteilte Evaluation für Fall 1)*

Wir geben einen verteilten Algorithmus an, der das Problem der Evaluation, wie wir es als Fall 1) für einen zentralen Nutzer spezifiziert haben, löst. Ausgehend von den Blättern des Spannbaums, schickt jeder Teilchecker nach und nach ein k-Tupel an seinen Elternknoten mit Ausnahme der Wurzel. Das k-Tupel eines Teilcheckers enthält dabei die Wahrheitswerte der entschiedenen lokalen Prädikate γ_j für seine Komponente.

Erhält ein Teilchecker solch ein k-Tupel von seinen Kindern, dann kombiniert er dieses mit seinem eigenem k-Tupel zu neuen Wahrheitswerten. Wenn das Verteilungsprädikat Γ_j universell-verteilbar ist, dann

bildet er die Konjunktion der Wahrheitswerte an der j-ten Stelle der beiden k-Tupel. Wenn das Verteilungsprädikat Γ_j hingegen existenziell-verteilbar ist, dann bildet er die Disjunktion der Wahrheitswerte an der j-ten der beiden k-Tupel.

Erst wenn ein Teilchecker die k-Tupel aller Kinder verarbeitet hat, schickt er sein k-Tupel an seinen Elternknoten. Auch die Wurzel verarbeitet die k-Tupel all ihrer Kinder entsprechend. Zusätzlich benutzt sie die so gewonnenen Wahrheitswerte und verknüpft sie logisch so, wie es der Aufbau des verteilbaren Zeugenprädikats für die Verteilungsprädikate vorgibt.

10.2.2.2 *Verteilte Evaluation für Fall 2)*

Ein verteilter Algorithmus an, der zusätzlich den Spannbaum benutzt, um von der Wurzel ausgehend alle Teilchecker zu informieren, löst das Problem der Evaluation, wie wir es als Fall 2) für einen verteilten Nutzer spezifiziert haben.

10.2.2.3 *Korrektheit der Evaluation*

Der vorgeschlagene Algorithmus zur Evaluation löst das Problem korrekt, denn es gilt:

Theorem 10.2.1 (Korrektheit der Evaluation)**.** *Nach der Ausführung der verteilten Evaluation für Fall 1) in einem Netzwerk kennt die Wurzel den Wahrheitswert des verteilbaren Zeugenprädikats.*

Den Beweis können wir mit einer Induktion über die Pfade des Spannbaums führen. Wir verzichten hier jedoch auf einen technischen Beweis. Dank des Spannbaums lässt sich auch die Terminierung leicht zeigen. Für den Fall 2) kommt dann lediglich noch die Bekanntmachung, also eine zweite Welle von Nachrichten im Spannbaum, hinzu.

10.3 EIGENSCHAFTEN DER EVALUATION

Wir betrachten in diesem Abschnitt einige Eigenschaften der Evaluation.

In Abschnitt 10.3.1 gleichen wir die Evaluation zunächst mit den allgemeinen Anforderungen an einen Checker ab. In Abschnitt 10.3.2 erläutern wir, warum für eine Evaluation ein Spannbaum oder eine äquivalente Kommunikationsstruktur zur Koordination im Netzwerk benötigt wird. In Abschnitt 10.3.3 zeigen wir Variationen der Evaluation für spezielle verteilbare Zeugenprädikate auf.

10.3.1 Abgleich mit den Anforderungen an einen Checker

Wir diskutieren, inwiefern die Evaluation die allgemeinen Anforderungen an einen Checker erfüllt.

10.3.1.1 *Verteiltheit, Gleichheit, Lokalität der Evaluation*

Ein verteilbares Zeugenprädikat haben wir so definiert, dass sich die Entscheidung durch die Summe der Entscheidungen der lokalen Prädikate ergibt. Bei der Evaluation werden lediglich Nachrichten mit Wahrheitswerten verschickt und verarbeitet. Die eigentliche Entscheidung des Zeugenprädikats wird durch lokale Prädikate gleichmäßig auf die Komponenten verteilt. Wir beachten damit also die Kerngedanken der Verteiltheit, Lokalität und Gleichheit.

Ein Nachteil ist hierbei die Koordination mithilfe eines Spannbaums. Die Wurzel übernimmt dabei als zusätzliche Berechnung die Auswertung für die Wahrheitswerte der Verteilungsprädikate. Dadurch ist die Gleichheit nicht vollständig erreicht. Dies trifft allerdings auf die meisten verteilten Algorithmen zu.

Die so arbeitenden Checker unterscheiden sich dennoch grundlegend von den skizzierten zentralen Checker (siehe Abschnitt 6.4 auf Seite 71). Während bei zentralen Checker die Teileingaben, Teilausgaben und potenziellen Teilzeugen eingesammelt und dann zentralisiert verifiziert werden, werden hier die Teilergebnisse der Entscheidung eingesammelt. Dabei wird die Entscheidung verteilt.

10.3.1.2 *Formale Verifikation*

Für eine formale Verifikation müssen die beiden Wellen der Kommunikation auf dem Spannbaum verifiziert werden: Eine Welle von den Blättern aus startend zur Evaluation und eine Welle von der Wurzel ausgehend zur Bekanntmachung des Ergebnis der Evaluation.

Die Kommunikation auf einem Spannbaum hat den Vorteil, dass die Korrektheit induktiv über die Pfade des Spannbaums gezeigt werden kann. Wir halten eine formale Verifikation der Evaluation daher für besonders gut handhabbar im Vergleich zu den üblichen verteilten Algorithmen der Literatur.

10.3.1.3 *Invasivität*

Wir argumentieren, dass die Evaluation nicht viele Ressourcen verbraucht und somit kaum invasiv ist. Bei einer Anzahl von n Komponenten im Netzwerk sind genau $n-1$ Nachrichten zur Evaluation und genauso viele zur Bekanntmachung nötig. Bei weniger Nachrichten wäre eine Komponente nicht beteiligt.

Die Evaluation ist außerdem gleichmäßig auf die Komponenten verteilt; es gibt keine Komponente, die einen ungewöhnlich hohen lokal Berechnungsaufwand hat, wie wir es für zentrale sequentielle Checker gesehen haben.

10.3.2 Evaluationsbaum

Für die verteilte Evaluation haben wir einen Spannbaum zur globalen Koordination in einem Netzwerk angenommen, den wir fortan *Evaluationsbaum* nennen. Tatsächlich ist eine globale Koordination auch notwendig für die Evaluation.

10.3.2.1 *Globale Koordination*

Mindestens eine Komponente des Netzwerks muss laut Spezifikation den Wahrheitswert des Zeugenprädikats kennen. Um diesen zu ermitteln, muss zumindest diese eine Komponente die Wahrheitswerte der lokalen Prädikate für alle Komponenten kennen. Daraus folgt die Notwendigkeit einer globale Koordination der Evaluation im Netzwerk. Hierfür kann ein Spannbaum eingesetzt werden.

Auch ein Koordinator würde ausreichen, da dieser einen Broadcast, zum Beispiel mit dem Echo-Algorithmus, initiieren kann. Der Echo-Algorithmus selbst baut allerdings auch einen Spannbaum auf und somit sind die beiden Varianten der globalen Koordination gleichwertig.

Da das Spannbaumproblem nicht lösbar in anonymen Netzwerken [Sano6], benötigen wir für die Evaluation ein ID-basiertes Netzwerk.

10.3.2.2 *Laufzeitverifikation: Spannbaum*

Die Korrektheit der globalen Koordination ist folglich Teil der Korrektheit eines Checkers. In Abschnitt 9.3 haben wir eine zertifizierende Spannbaumkonstruktion eingeführt. Wir können diese jedoch nicht benutzen, um die Korrektheit zu garantieren, da wir hierbei in einen Zirkelschluss geraten würden.

Wir haben durch einen korrekten verteilten Checker also auch immer einen korrekten Spannbaum gegeben. Eine zertifizierende Spannbaumkonstruktion ist dennoch interessant, da Spannbäume bei verteilten Algorithmen eine große Rolle spielen und ein zur Laufzeit benötigter Spannbaum nicht zwangsläufig dem Evaluationsbaum entsprechen muss.

10.3.3 Variationen der Evaluation

Wir betrachten Variationen der Evaluation.

10.3.3.1 *Zählen bei der Evaluation*

Wir haben verteilbare Prädikate so definiert, dass eine Verschachtelung von Quantoren nicht möglich ist (siehe Abschnitt 6.5). Diese Einschränkung hat den Vorteil, dass die Kommunikation während der Evaluation eines beliebigen verteilbaren Zeugenprädikats immer gleich strukturiert ist.

Mit einer geringfügigen Anpassung der Evaluation sind jedoch Quantoren für die gezählt werden muss integrierbar. Ein Beispiel stellt ein Quantor dar, der beschreibt, dass eine bestimmte Anzahl an Komponenten eine Eigenschaft erfüllt.

10.3.3.2 *Praktische Lösung: Time-Out*

In der Literatur zur Laufzeitverifikation von Netzwerken (insbesondere dem Monitoring) wird häufig von einem Alarmschlagen der Komponenten gesprochen [FPS18]. Das heißt, es wird eine Sicherheitseigenschaft zur Laufzeit geprüft und wenn es nach einer bestimmten Zeit, dem *Time-Out*, keinen Alarm gibt, dann wird davon ausgegangen, dass die Sicherheitseigenschaft gilt.

In unserem Fall könnte ein Time-Out anstelle einer Evaluation für ein universell-verteilbares Zeugenprädikat eine praktische Lösung sein. Eine Komponente deren lokales Prädikat nicht erfüllt ist, schlägt Alarm. Es ist jedoch eine praktische Lösung, die auch einen Preis hat.

In einem asynchronen Netzwerk können die Komponenten nie sicher sein, dass sie lange genug gewartet haben. Es kann deswegen falsche Positive gebe: ein Checker akzeptiert, aber das Eingabe-Ausgabe-Paar ist nicht korrekt. Dies verletzt das Konzept eines zertifizierenden Algorithmus. Ein Time-Out kann deswegen eine praktische Lösung sein, aber es kann die Evaluation im Allgemeinen nicht ersetzen.

Für ein synchrones System kann ein Time-Out jedoch die Evaluation universell-verteilbarer Zeugenprädikate ersetzen.

10.4 PRÜFUNG DER KONSISTENZ VERTEILTER ZEUGEN

Ein Zeuge ist ein Korrektheitsargument; es ist deswegen notwendig, dass der Zeuge konsistent ist (siehe Kapitel 7). Wir betrachten

in diesem Abschnitt die Prüfung der Konsistenz verteilter Zeugen. Ergebnisse dieses Abschnitts haben wir in [VA18] veröffentlicht.

In Abschnitt 10.4.1 diskutieren wir die Prüfung für beliebige Zeugen und in Abschnitt 10.4.2 die Prüfung für zusammenhängende Zeugen. In Abschnitt 10.4.3 vergleichen wir die Varianten der Konsistenzprüfung abschließend.

10.4.1 Konsistenzprüfung für beliebige Zeugen

Ein Zeuge ist konsistent, wenn seine Teilzeugen paarweise konsistent sind (siehe Lemma 7.3.1, Abschnitt 7.3).

10.4.1.1 *Paarweise Prüfung*

Eine Möglichkeit der Konsistenzprüfung ist deswegen der paarweise Vergleich aller Teilzeugen. Ein Teilchecker muss dafür seinen Teilzeugen an alle anderen Teilchecker schicken.

Die Konsistenzprüfung ist damit offensichtlich aufwändig; sie kann mithilfe eines Spannbaums besser strukturiert werden.

10.4.1.2 *Prüfung mithilfe eines Spannbaums*

Hierbei ist eine Möglichkeit, dass alle Teilchecker ihre Teilzeugen im Spannbaum an ihre Eltern hoch reichen und die Wurzel alle Vergleiche der Konsistenzprüfung übernimmt. Die Prüfung ist dabei sehr ungleich verteilt, da die sie vollständig in der Wurzel stattfindet.

Eine Optimierung hierfür ist, dass beim Hochreichen bereits geprüft wird. Erhält ein Teilchecker Teilzeugen von seinen Kindern, so vergleicht er alle gemeinsamen Variablen auf konsistente Belegungen. Er führt die Teilzeugen seiner Kinder und seinen eigenen zu einem neuen Teilzeugen zusammen.

Im Falle der Konsistenz enthält dieser Teilzeuge jede Variable mit entsprechender Belegung einmal. Im Falle einer Inkonsistenz enthält der Teilzeuge dafür nur einen speziellen Marker. Erhält ein Teilchecker einen Teilzeugen mit einem solchen Marker, dann entfällt eine weitere Prüfung und er reicht ihn an seinen Elternknoten weiter.

10.4.1.3 *Prüfung der Wohlgeformtheit*

Wir haben uns auf wohlgeformte Zeugen beschränkt, damit wir die Konsistenz nur für Zeugen, nicht aber für Eingabe und Ausgabe betrachten müssen. Wenn der Teilzeuge einer Komponente alle Informationen der Teileingabe und der Teilausgabe enthält, ist der Zeuge wohlgeformt (siehe Lemma 7.3.5).

Jeder Teilchecker kann die Wohlgeformtheit des Teilzeugen lokal für seine Komponente prüfen. Selbiges könnte jedoch auch durch ein Typsystem garantiert sein. Wir haben in Kapitel 7 erläutert, dass es sich bei der Wohlgeformtheit um eine technische Lösung handelt, um hier die Betrachtungen zur Konsistenz einfach zu halten. Im weiteren gehen wir deswegen nicht auf die Wohlgeformtheit ein.

10.4.2 Konsistenzprüfung für zusammenhängende Zeugen

Zusammenhängende Zeugen sind deshalb interessant, weil ihre Konsistenzprüfung mit einem *lokalen* verteilten Algortihmus gelingt.

10.4.2.1 *Lokale Konsistenzprüfung*

Die Idee für die Konsistenzprüfung ergibt sich aus dem Theorem 31. Für die Konsistenz eines zusammenhängenden Zeugen ist es ausreichend, wenn alle Nachbarn paarweise konsistente Teilzeugen besitzen. Deswegen muss der Teilchecker einer Komponente lediglich die 1-Nachbarschaft betrachten. Jeder Teilchecker prüft die Konsistenz seines Teilzeugen mit den Teilzeugen benachbarter Teilchecker. Somit ist die Konsistenzprüfung lokal.

Alle in dieser Arbeit aufgezeigten Zeugen sind zusammenhängend. Mit dem Lemma 7.3.3 haben wir außerdem gezeigt, dass wenn es einen Zeugen für ein Eingabe-Ausgabe-Paar gibt, dann gibt es auch immer einen zusammenhängenden Zeugen für dasselbe Eingabe-Ausgabe-Paar. Der zusammenhängende Zeuge ergibt sich dabei kanonisch aus dem nicht-zusammenhängenden.

Der Zusammenhang eines Zeugen ist notwendige Voraussetzung dafür, dass die lokale Konsistenzprüfung gelingt. Das bedeutet, dass ein Checker auch den Zusammenhang prüfen muss.

Die Prüfung der Konsistenz und die des Zusammenhangs untrscheiden sich jedoch grundlegend von einander. Die Konsistenz von der Belegung der Variablen abhängig und der Zusammenhang nicht. Deshalb muss die Konsistenz zur Laufzeit für jede Ausführung geprüft werden.

Der Zusammenhang hingegen hängt davon ab, wie die Form der Teilzeugen aussieht, also für welche Variablen Werte berechnet werden. In einem Netzwerk, in dem ein zertifizierender verteilter Algorithmus mehrmals ausgeführt wird, genügt es den Zusammenhang einmalig zu prüfen. Die Teilchecker müssen danach lediglich sicherstellen, dass der Teilzeuge auch bei jeder weiteren Ausführung die gleichen Variablen enthält.

10.4.2.2 *Prüfung des Zusammenhangs*

Der Zusammenhang kann mit einer verteilten Variante eines Graphalgorithmus berechnet werden [HN19]. Ein Netzwerk hängt immer zusammen, deswegen ist eine verteilte Berechnung von Zusammenhangskomponenten nicht üblich. Hier geht es jedoch um die Zusammenhangskomponenten, die jeweils durch die Variablen der Teilzeugen entstehen (siehe Abschnitt 7.3).

Die Idee hierbei ist, dass für jede Variable α eines Teilzeugen die zusammenhängenden Teilgraphen berechnet werden, die die α-Komponenten bilden. Für jede Zusammenhangskomponente der α-Komponenten wird ein Leader gewählt, zum Beispiel die α-Komponente mit kleinster ID.

Dabei startet jeder Teilchecker mit seiner Komponente als Leader für all die Variablen des Teilzeugen und sendet seinen Vorschlag an seine Nachbarn. Enthält ein Teilchecker einen neuen Vorschlag für eine seiner Variabelen, dann sendet er den Vorschlag weiter an seine Nachbarn. Ist der Vorschlag außerdem besser, so aktualisiert er den Leader für diese Variable. Erhält ein Teilchecker einen Vorschlag für eine Variable, die nicht in dem Teilzeugen vorkommt, so sendet er sie nicht weiter. So entstehen Zusammenhangskomponenten über den Variablen der Teilzeugen.

Ist die Wahl des Leaders abgeschlossen, so besitzt jeder Teilchecker eine Liste, die jede Variable des eigenen Teilzeugen mit der entsprechenden ID des jeweiligen Leaders assoziiert. Die Liste wird über den Evaluationsbaum hoch gereicht. Die Wurzel prüft, dass es für keine Variable mehrere Leader gibt.

10.4.3 Vergleich

Wir vergleichen die Konsistenzprüfung beliebiger Zeugen mit der für zusammenhängende Zeugen. Der entscheidende Vorteil der Konsistenzprüfung für beliebige Zeugen ist, dass es keine weiteren Anforderungen an einen Zeugen gibt.

Sie hat jedoch den Nachteil, dass Teilzeugen durchs Netzwerk geschickt werden. Dadurch werden zum einen Informationen im gesamten Netzwerk geteilt und zum anderen gibt es mehr Nachrichtenverkehr und das mit eventuell großen Nachrichten. Die Größe eines Teilzeugen hängt von dem zertifizierenden verteilten Algorithmus ab. Ein weiterer Nachteil ist, dass die Prüfung an der Wurzel zentralisiert wird. Die beschriebene Optimierung verschafft hier Abhilfe, aber am Worst-Case-Szenario ändert sich nichts.

Der Nachteil einer lokalen Konsistenzprüfung ist entsprechend die Notwendigkeit der Voraussetzung eines Zusammenhangs eines Zeu-

gen. Abhilfe schafft hier, dass der Zusammenhang generalisierbar ist und nicht für jede Ausführung geprüft werden muss.

Der entscheidende Vorteil der Konsistenzprüfung zusammenhängender Zeugen ist ihre Lokalität. Damit verbundenen sind eine geringe Nachrichtenkomplexität, sowie die Verteiltheit der Prüfung. Darüber hinaus beobachten wir, dass diese Prüfung auch für nicht-terminierende verteilte Algorithmen skaliert. Wir betrachten hier zwar die Zertifizierung terminierender verteilter Algorithmen, wir können uns dennoch leicht vorstellen, dass ein Checker bei einer Zertifizierung bei Nicht-Terminierung kontinuierlich immer wieder prüft. Somit ist die aufwändigere, aber einmalige Prüfung des Zusammenhangs und die lokale Prüfung der Konsistenz besonders interessant.

11 INDUSTRIELLE FALLSTUDIE: VIRTUELLE NETZWERKE ZUR REDUKTION DER INVASIVITÄT EINES CHECKERS

Wir stellen in diesem Kapitel eine industrielle Fallstudie vor, die wir in [AV19] veröffentlicht haben. Die Fallstudie ist ermöglicht worden durch das CrESt-Projekt [1] gefördert durch das Bundesministeriums für Bildung und Forschung, in dessen Rahmen der Projektpartner InSystems [Ins] einen Anwendungsfall bereitstellt.

In Abschnitt 11.1 stellen wir in Kürze das Projekt und insbesondere den Projektpartner InSystems, sowie die Motivation für den Einsatz zertifizierender verteilter Algorithmen vor.

In Abschnitt 11.2 präsentieren wir die Fallstudie, eine zertifizierende verteilte Auktion für ein Multi-Agenten-System. Interessant ist hierbei, dass wir die Welt der Netzwerke verlassen und mit einem Multi-Agenten-System auch eine andere Klasse verteilter Systeme betrachten. Das hat insbesondere einen Einfluss auf den Checker und dessen Invasivität.

In Abschnitt 11.3 betrachten wir deswegen den verteilten Checker, sowie eine Reduktion der Invasivität durch Alternativen bei der Kommunikation der Teilchecker durch *virtuelle Netzwerke* – ein virtuelles Netzwerk gibt auf einem verteilten System eine Struktur für die Kommunikation vor. Perspektivisch ist diese Reduktion auch für andere Fallstudien zur Zertifizierung in verteilten Systemen relevant ist.

11.1 CREST-PROJEKT: EINSATZ ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Im Rahmen des Projekts „**C**ollabo**r**ative **E**mbedded **S**yst**e**ms“ (CrESt) stellt der Projektpartner InSystems eine Auswahl textuell-beschriebener Anwendungsfälle bereit.

1 Projektseite: https://crest.in.tum.de/

In Abschnitt 11.1.1 führen wir das CrESt-Projekt und den Projektpartner InSystems ein. In Abschnitt 11.1.2 diskutieren wir die im Rahmen des Projekts herausgestellten Herausforderungen und in Abschnitt 11.1.3 motivieren wir schließlich den Einsatz zertifizierender verteilter Algorithmen als eine Methode mit den Herausforderungen umzugehen. In Abschnitt 11.1.4 zeigen wir dann auf, wie der Einsatz gelingt.

11.1.1 CrESt-Projekt

Wir führen in Kürze das CrESt-Projekt und insbesondere den Industriepartner InSystems ein.

11.1.1.1 *Ziel des Projekts*

In dem CrESt-Projekt geht es um kollaborierende eingebettete Systeme. Auf der Projektseite wird das Ziel des Projekts, wie folgt beschrieben: [2]

> ein umfassendes Rahmenwerk für die Entwicklung kollaborierender eingebetteter Systeme zu schaffen, das die neuartigen Herausforderungen in der Entwicklung kollaborierender eingebetteter Systeme in dynamischen Systemverbünden [...] adressiert [...].

Wir setzen die Betrachtung kollaborierender eingebetteter Systeme in den Kontext zu unseren Betrachtungen zu Netzwerken, nachdem wir den Einsatz zertifizierender verteilter Algorithmen motivieren (siehe Abschnitt 11.1.4). Bis dahin appellieren wir an die Intuition und Geduld der Leser:innen. Wir möchten zunächst einen groben Überblick vermitteln.

11.1.1.2 *Industriepartner* InSystems

Das CrESt-Projekt hat über zwanzig Partner, die in etwa zur Hälfte aus akademischen Partnern und zur anderen Hälfte aus Industriepartnern besteht. Die Firma InSystems ist einer der Industriepartner und entwickelt Transportroboter, die sowohl autonom agieren als auch kollaborieren.

Die Robotern werden zum Beispiel eingesetzt, um Maschinen zu bedienen, indem sie benötigte Rohstoffe und ähnliches zur entsprechenden Maschine transportieren. Die Abbildung 11.1 zeigt Transportroboter im Einsatz in einer Fabrik.

2 Projektseite: https://crest.in.tum.de/

Abbildung 11.1: Transportroboter in einer Fabrik. Die Abbildung stammt aus [Sch18].

11.1.1.3 *Technische Details*

Eine Flotte besteht typischerweise aus vier bis acht Robotern, die jeweils eine Transportkapazität von 50-200 kg besitzen und sich mit einer Geschwindigkeit von 0,5-2 m/s bewegen [Sch18]. Jeder Roboter arbeitet autonom, das heißt er entscheidet welche Transportaufgaben er übernimmt, welchen Wege er zurücklegt, wann er seine Batterien auflädt und vieles mehr.

11.1.1.4 *Ziele von* InSystems

Dabei soll es keine zentrale Steuerung geben. Innerhalb der Flotte soll jeder Roboter seine eigenen Ziele verfolgen, wie zum Beispiel seine Batterie stetig geladen zu halten oder den kürzesten Weg zwischen zwei Orten zurückzulegen. Zusätzlich soll die gesamte Flotte an Robotern auch gemeinsame Ziele verfolgen, wie zum Beispiel dass die Maschinen der Fabrik möglichst schnell bedient werden, dass jede Maschine der Fabrik ausgelastet ist oder die anfallenden Transportaufgaben innerhalb der Flotte optimal verteilt werden.

11.1.1.5 *Kollaboration der Roboter*

Jeder Roboter ist für die Verwirklichung seiner eigenen Ziele verantwortlich. Für die gemeinsamen Ziele ist die Flotte an Robotern jedoch gemeinsam verantwortlich; um die gemeinsamen Ziele zu erreichen, kollaborieren die Roboter miteinander.

Die Basis für die Kollaboration bilden dabei verteilte Algorithmen. Die Roboter kommunizieren untereinander über WLAN. Jeder Roboter

hat eine eindeutige ID, die zunächst nur er selbst kennt. Ein Roboter kann eine Nachricht an einen speziellen Roboter verschicken, wenn er dessen ID kennt.

Darüber hinaus kann jeder Roboter eine Broadcast-Nachricht schicken, also eine Nachricht, die an alle Roboter der Flotte verschickt wird. Hierfür benötigt ein Roboter keine IDs der Empfänger. Jede Maschine der Fabrik kann außerdem auch eine Broadcast-Nachricht an die Flotte senden, um unter anderem mitzuteilen, dass sie bedient werden soll, also einen Transportauftrag für die Flotte hat.

11.1.2 Herausgestellte Herausforderungen des Fraunhofer Instituts FOKUS

Wir beschreiben die vom Projektpartner, Fraunhofer Institut für offene Kommunikationssysteme, herausgestellten Herausforderungen. [3] Insbesondere die Notwendigkeit einer zentralen Steuerung und einer bekannten Umgebung seien Herausforderungen kollaborierender eingebetteter Systeme:

> Die existierende Softwaretechnologie ist abhängig von einer zentralen Steuerung und ermöglicht den Einsatz der Systeme nur in vollständig bekannten und stabilen Umgebungen.

Ein Ziel sei deswegen, eine dezentrale Steuerung der Flotte; das heißt die Autonomie der Roboter wird erhöht. Weiteres Ziel sei es, den Einsatz in einer unbekannten Umgebung zu ermöglichen:

> Im Rahmen des CrESt-Projekts arbeiten die Projektpartner an Konzepten und Werkzeugen, um eingebettete Systeme zur intelligenten und autonomen Kollaboration zu befähigen. Weiterhin sollen die Architekturen der Systeme so weiterentwickelt werden, dass ein Einsatz auch in teilweise unbekannten Umgebungen möglich ist.

Diese Ziele sollen sich entsprechend auch in der Wahl der Methoden zur Qualitätssicherung kollaborierender eingebetteter Systeme widerspiegeln.

11.1.3 Motivation für den Einsatz zertifizierender verteilter Algorithmen

Wir argumentieren, dass sich zertifizierende verteilte Algorithmen für den Einsatz gut eignen.

3 Die Projektseite des CrESt-Projekts des Projektpartners Fraunhofer Institut für offene Kommunikationssysteme ist https://www.fokus.fraunhofer.de/de/sqc/news/crest.

11.1.3.1 *Dezentrale Steuerung*

Verteilte Algorithmen bieten die Grundlage für die Kollaboration der Roboter innerhalb ihrer Flotte. Für die Laufzeitverifikation mit zertifizierenden verteilten Algorithmen ist keine zentrale Steuerung nötig. Der Grund dafür ist deren dezentralisierte Verifikation. Ein Zeuge wird verteilt berechnet, indem jede Komponente ihren Teilzeugen berechnet.

Ein Checker eines verteilbaren Zeugenprädikats kommt außerdem fast ohne Zentralisierung aus. Wir benötigen lediglich einen Evaluationsbaum. Ein Spannbaum für die Evaluation bringt nur einen geringen Grad an Zentralisierung in das System, da die Entscheidung des verteilbaren Zeugenprädikats auf die Komponenten des Systems verteilt wird. Wir ändern durch einen Evaluationsbaum auch nicht die grundlegenden Voraussetzungen des Systems, da es sich um ein ID-basiertes System handelt.

11.1.3.2 *Unbekannte Umgebung*

Die Verifikation findet beim Einsatz zertifizierender verteilter Algorithmen zur Laufzeit statt. Die Umgebung muss für die Verifikation deswegen nicht schon zur Design-Zeit bekannt sein. Es ist zum Beispiel nicht nötig für die Zertifizierung die Gesamtzahl der Roboter von vornherein zu kennen.

11.1.4 Einsatz zertifizierender verteilter Algorithmen

Wir haben bisher Netzwerke betrachtet. Bei der Roboterflotte handelt es sich jedoch um ein Multi-Agenten-System [DKJ18]. Wir zeigen deswegen auf, wie sich die neuen Konzepten zu unseren bisherigen für den Einsatz zertifizierender verteilter Algorithmen verhalten.

11.1.4.1 *Roboterflotte als Multi-Agenten-System*

Ein *Multi-Agenten-System* ist ein ein verteiltes System, bei dem jede Komponente ein *Agent* ist; das heißt, er ist eine autonom agierende rechnende Einheit.

Bei unserer abstrakten Betrachtung von Netzwerken gehen wir bei Komponenten von beliebigen rechnenden Einheiten aus. Das heißt, dass eine Komponente eines Netzwerks durchaus auch ein Agent sein kann. Wenn man von einem Multi-Agenten-System spricht, betont man den Aspekt, dass ein Agent in einem solchen System individuelle Ziele verfolgt. Die individuellen Ziele der Agenten können dabei auch in Konflikt geraten mit den gemeinsam verfolgten Zielen im Multi-Agenten-System.

11.1.4.2 *Teilchecker als Thread*

In der vorliegenden Fallstudie ist solch ein Agent ein Roboter. Ein Roboter ist dabei selbst wieder ein verteiltes System, denn verschiedene Threads rechnen teilweise unabhängig von einander auf dem Roboter und kommunizieren dabei auch untereinander, um die individuellen Ziele eines Roboters umzusetzen. Es ist deswegen kein Problem jeden Roboter weiterhin mit einem Teilchecker auszustatten. Der Teilchecker ist ein Thread, der auf dem jeweiligen Roboter läuft.

11.1.4.3 *Roboter als eingebettetes System*

Ein Roboter ist außerdem auch ein eingebettetes System [Whi11]. Ein *eingebettetes System* ist eine rechnende Einheit, die in einen Kontext, hier die Fabrik, eingebunden ist und dabei spezielle Aufgaben übernimmt, hier zum Beispiel Transportaufgaben. Klassischerweise ist die Hardware eines eingebetteten Systems dabei auf die speziellen Aufgaben ausgerichtet. Dies gilt auch für die Transportroboter.

Für unsere Betrachtungen ist ein Roboter abstrakt eine Komponente eines verteilten Systems.

11.1.4.4 *Topologie: vollständiger Graph*

Wir fassen das verteilte System der Roboterflotte außerdem als ein spezielles Netzwerk auf. Während wir für ein beliebiges Netzwerk nicht wissen, wie dessen Topologie aussieht, wissen wir für die Roboterflotte mehr. Eine Broadcast-Kommunikation ist Teil des Systems und somit jeder Roboter mit jedem anderen Roboter benachbart. Wir erachten deswegen eine Betrachtung als sinnvoll, bei der wir die Roboterflotte als ein Netzwerk sehen, dessen Topologie ein vollständiger Graph ist.

11.1.4.5 *Kollaboration*

Um gemeinsame Ziele zu verfolgen, bedienen sich die Roboter verteilter Algorithmen. Es sind verteilte Algorithmen, die den Robotern die Kollaboration ermöglichen. Genau auf den verteilten Algorithmen liegt der Fokus der vorliegenden Fallstudie.

11.2 ZERTIFIZIERENDE VERTEILTE AUKTION

Als Fallstudie betrachten wir die verteilte Auktion. Ein verteilter Algorithmus zur Kollaboration, der ausgeführt wird, wenn eine Maschine

einen Auftrag, zum Beispiel einen Transportauftrag, zu vergeben hat. Durch die verteilte Auktion wird innerhalb der Flotte entschieden, welcher Roboter den Auftrag übernimmt.

In Abschnitt 11.2.1 stellen wir die verteilte Auktion vor. In Abschnitt 11.2.2 geben wir eine Spezifikation für die verteilte Auktion an, die Grundlage für deren Zertifizierung ist. In Abschnitt 11.2.3 beschreiben wir ein verteilbares Zeugenprädikat für die verteilte Auktion.

11.2.1 Verteilte Auktion

Der verteilte Algorithmus zur Auktion läuft wie folgt ab [Ins]. Eine der Maschinen hat einen neuen Auftrag zu vergeben. Sie sendet deswegen eine Broadcast-Nachricht mit einer Auftrags-ID an alle Roboter. Damit startet die verteilte Auktion für den neuen Auftrag.

Jeder Roboter berechnet individuell sein Gebot für den neuen Auftrag und sendet seine ID und die Auftrags-ID als Broadcast-Nachricht an die Roboterflotte. Jeder Roboter berechnet sein Gewinner-Tripel *(Gewinner-Gebot, Gewinner-ID, Auftrags-ID)*, indem er das Maximum aller Gebote als *Gewinner-Gebot* wählt und die zugehörige ID als *Gewinner-ID* auswählt.

Das Gebot berechnet ein Roboter durch einen sequentiellen Algorithmus, der von den individuellen Zielen des Roboters abhängen könnte, wie zum Beispiel seiner Notwendigkeit seine Batterie aufzuladen. Er könnte aber auch Ziele der Flotte berücksichtigen, wie dass ein möglichst nah stehender Roboter die Maschine bedient. Der konkrete Algorithmus liegt außerhalb unserer Betrachtung und ist für die Zertifizierung irrelevant.

11.2.1.1 *Symmetrie brechen*

Bei mehreren maximalen Geboten müssen die Roboter die Symmetrie brechen, um einen Gewinner zu bestimmen. Der Algorithmus gibt hier keine Lösung vor, die Symmetrie kann jedoch beispielsweise mithilfe der IDs der Roboter gebrochen werden. Dabei nehmen wir entsprechend eine Ordnung auf den IDs an.

11.2.1.2 *Time-Out*

Ein weiteres Problem ist, dass ein Roboter nicht weiß, wann er alle Gebote gesehen hat, da er die Größe der Flotte nicht kennt. Als Lösung für dieses Problem sieht InSystems ein Time-Out vor. Folglich kann es passieren, dass ein Roboter sein Gewinner-Tripel berechnet ohne alle Gebote gesehen zu haben, falls das Time-Out zu gering angesetzt ist.

Wir lösen das Problem so, dass sich ein Roboter während der Laufzeitverifikation beschweren kann, wenn sein Gebot nicht beachtet wurde und er der eigentliche Gewinner gewesen wäre.

In [AL19] wird auf die hier beschriebene Fallstudie aufgebaut. Dabei werden verschiedene Methoden kombiniert, um neben funktionalen Eigenschaften zum Beispiel auch zeitlichen Aspekten, wie das Einhalten von Deadlines zu gewährleisten.

11.2.1.3 *Optionale Teilnahme an der Auktion*

InSystems sieht vor, dass jeder Roboter autonom entscheiden kann, ob er überhaupt an der Auktion teilnimmt. Diese Wahl wird jedoch nicht im vorliegenden Algorithmus beachtet. Des Weiteren liegt uns keine Spezifikation dafür vor, was in dem Extremfall, dass kein Roboter teilnimmt, passieren soll.

Dieser Fall ist insofern problematisch, da das am höchsten priorisierte Ziel der Flotte das Bedienen der Maschinen ist. Wir gehen deswegen im folgenden zur Vereinfachung davon aus, dass all Roboter teilnehmen.

Die Abbildung 11.2 zeigt die verteilte Auktion als ein UML-Diagramms (Unified Modelling Language) [Fow03], genauer gesagt in einer für Variante eines UML-Diagramms, das für Agenten-Systeme erweitert wurde [Hug02]. Dabei sind auch ein Time-Out für die Abgabe eines Gebots und eine optionale Teilnahme an der Auktion dargestellt. Die Abbildung 11.2 dient einer besseren Intuition, die im folgenden Abschnitt gegebene Spezifikation bildet jedoch die Grundlage für die Laufzeitverifikation.

11.2.2 Spezifikation verteilte Auktion

Als Grundlage für die Laufzeitverifikation mithilfe einer zertifizierenden Variante der verteilten Auktion dient uns die folgende Spezifikation. Die verteilte Auktion für einen Auftrag ist korrekt, wenn die folgenden Nachbedingungen gelten:

Einigkeit: Alle Roboter sind sich einig über die Gewinner-ID und die Auftrags-ID.

Existenz: Es existiert genau ein Roboter dessen ID der Gewinner-ID entspricht.

Maximum: Das Gewinner-Gebot ist ein Maximum bezüglich aller Gebote für den Auftrag.

In einem anderen Kontext würde die Nachbedingung zum Maximum zur Beschreibung der Funktionsweise des verteilten Algorithmus zählen und nicht zu dessen Spezifikation. Zum Beispiel hat bei der Leader

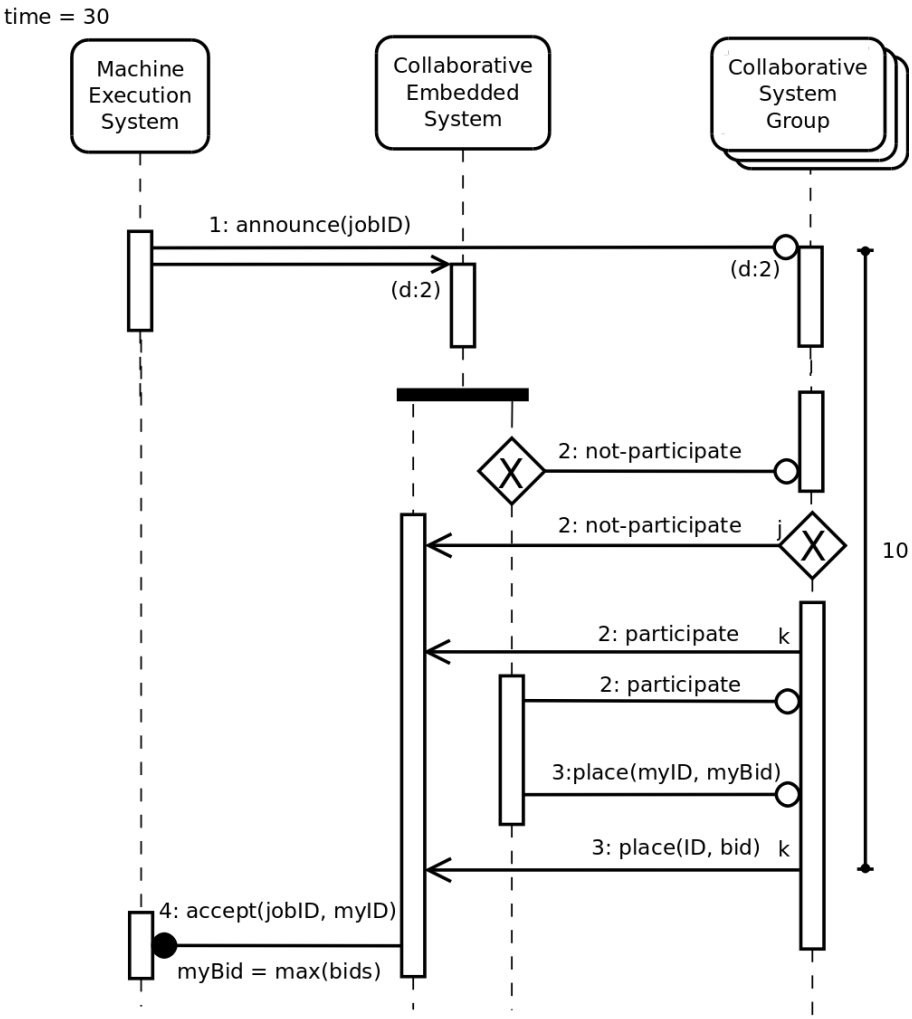


Abbildung 11.2: Verteilte Auktion als UML-Diagramm dargestellt. Die Abbildung stammt aus [AL19]. Ein Roboter ist dabei ein kollaboratives eingebettetes System, während die gesamte Flotte der Roboter eine kollaborative Systemgruppe ist.

Election (siehe Abschnitt 9.7) das Kriterium zur Wahl des Leaders keine Rolle gespielt.

Für die verteilte Auktion ist die Nachbedingung zum Maximum jedoch eine funktionale Eigenschaft. Ein Roboter gibt sein Gebot in Abhängigkeit seiner eigenen Ziele und der gemeinsamen Ziele ab. Für die Korrektheit des Multi-Agenten-Systems ist es deswegen unerlässlich, dass die Gebote berücksichtigt werden.

11.2.3 Verteilbares Zeugenprädikat

Wir geben verteilbare Prädikate an, deren Konjunktion die Nachbedingungen (Einigkeit, Existenz und Maximum) impliziert.

11.2.3.1 *Teileingaben und Teilausgaben*

Hierfür benötigen wir zunächst einmal die Teileingaben, sowie die Teilausgaben einer verteilten Auktion. Die Teileingabe eines Roboters v ist seine ID $\mathtt{id}_v$ und sein, von ihm bestimmtes, Gebot $\mathtt{bid}_v$. Die Teilausgabe eines Roboters v ist die ID $winnerID_v$ des von ihm bestimmten Gewinners der Auktion, sowie die Auftrags-ID $\mathtt{jobID}_v$ von der v ausgeht.

Für die Teilzeugen benötigen wir weiterhin das Gebot des gewählten Gewinners $winnerBid_v$. Während der verteilten Auktion erhält jeder Roboter die Gebote der anderen Roboter. Die Berechnung der Teilzeugen ist deswegen leicht in die verteilte Auktion integrierbar. Der Teilzeuge eines Roboters besteht darüber hinaus aus den Teilausgaben der Nachbarn.

Wir bezeichnen das Tripel $(winnerID_v, winnerBid_v, \mathtt{jobID}_v)$ als das Gewinner-Tripel von v.

11.2.3.2 *Verteilungsprädikate*

Wir geben nachfolgend die Verteilungsprädikate für die Nachbedingungen an.

Für die Nachbedingung der *Einigkeit* geben wir ein universell-verteilbares Prädikat Γ_{agree} an; das zugehörige lokale Prädikat γ_{agree} ist für einen Roboter v erfüllt, falls dessen Teilausgabe $(winnerID_v, \mathtt{jobID}_v)$ der Teilausgabe aller Nachbarn entspricht.

Für die Nachbedingung der *Existenz* geben wir ein existenziell-verteilbares Prädikat Γ_{ex} an; das zugehörige lokale Prädikat γ_{ex} ist für einen Roboter v erfüllt, falls

- die ID des von ihm gewählte Gewinner $winnerID_v$ seiner eigenen ID $\mathtt{id}_v$ entspricht und
- sein Gewinner-Gebot $winnerBid_v$ sein eigenes Gebot $\mathtt{bid}_v$ ist.

Wir beobachten, dass das Prädikat Γ_{ex} auch dann erfüllt ist, wenn es mehrere Gewinner gibt. Das lokale Prädikat reicht also noch nicht aus. Allerdings ist in Verbindung mit dem Prädikat für die Einigkeit Γ_{agree} garantiert, dass es höchstens einen Gewinner gibt.

Für die Nachbedingung, dass es sich um das *Maximum* handelt beim Gebot des Gewinners, geben wir ein universell-verteilbares Prädikat Γ_{max} an; das zugehörige lokale Prädikat γ_{max} ist für einen Roboter v erfüllt, falls sein eigenen Gebot $\mathtt{bid}_v$ kleiner oder gleich dem von ihm gewählten Gewinner-Gebot $winnerBid_v$ ist.

Wir beobachten, dass das Prädikat Γ_{max} auch dann erfüllt sein kann, wenn jeder Roboter ein anderes Gewinner-Gebot hat, mit dem er vergleicht. Allerdings ist in Verbindung mit dem Prädikat für die

Einigkeit Γ_{agree} dann garantiert, dass es höchstens einen Gewinner gibt.

Die Konjunktion der Verteilungsprädikate Γ_{ex}, Γ_{agree} und Γ_{max} impliziert also, dass die Spezifikation der verteilten Auktion erfüllt ist.

11.3 CHECKER

Wir präsentieren in diesem Abschnitt einen Checker für das beschriebene verteilbare Zeugenprädikat der verteilten Auktion. In Kapitel 10 haben wir die Architektur für Checker beschrieben. Wir erinnern uns daran, dass dabei jeder Teilchecker mit allen benachbarten Teilcheckern kommuniziert. Für den speziellen Fall eines Multi-Agenten-Systems, in welchem jede Komponente und somit jeder Teilchecker mit jedem anderen anderen Teilchecker benachbart ist, sind deswegen Alternativen für die Kommunikation von Teilcheckern sinnvoll, um die Invasivität des Checkers zu reduzieren.

In Abschnitt 11.3.1 führen wir deswegen Metriken für die Invasivität ein. In Abschnitt 11.3.2 stellen wir den generischen Teilchecker eines jeden Roboters vor. Auf Grundlage der Metriken zur Invasivität vergleichen wir in Abschnitt 11.3.3 Alternativen für die Kommunikation der Teilchecker hinsichtlich ihrer Invasivität. Dafür betrachten wir virtuelle Netzwerke, die eine Kommunikationsstruktur vorgeben.

11.3.1 Metriken für Invasivität

Als Kriterien für die Invasivität eines Checkers betrachten wir für diese Fallstudie die Nachrichtenkomplexität, die Laufzeit und die lokale Laufzeit der Teilchecker. Wir benutzen entsprechende Metriken für diese Kriterien.

Die *Nachrichtenkomplexität* bestimmt sich hierbei durch die empfangenen Nachrichten. Bei einer Broadcast-Nachricht gibt es deswegen so viele empfangene Nachrichten, wie es Roboter in der Flotte gibt. Wir drücken damit also auch den Overhead aus, der durch die Verarbeitung einer Nachricht entsteht. Wir lassen hier außer Acht, wie groß Nachrichten sein dürfen sind. In einem realen Netzwerk sind dadurch unter Umständen mehr Nachrichten nötig. Das gilt dann jedoch für alle Varianten in unseren Vergleichen.

Wie für asynchrone Netzwerke üblich, bestimmen wir die *Laufzeit* unter der Annahmen, dass das Senden einer Nachricht einer Zeiteinheit entspricht [Peloo].

Die *lokale Laufzeit* ergibt sich aus der sequentiellen Laufzeit eines Roboters. Es ist üblich bei der Analyse verteilter Algorithmen, die lokale

Laufzeit wegzulassen, sofern sie sich in einem vernünftigen Rahmen bewegt. Handelt es sich bei der lokalen Berechnung jedoch eigentlich um eine globale Berechnung (das heißt, die lokale Laufzeit ist von der Anzahl der Komponenten abhängig), dann wird meist darauf hingewiesen [Peloo]. Da es in unserer Fallstudie auch zu globalen Berechnungen kommt, betrachten wir die lokale Laufzeit explizit mit.

11.3.2 Teilchecker

Der Teilchecker eines Roboter ergibt sich aus den beschriebenen lokalen Prädikaten γ_{agree}, γ_{ex} und γ_{max} relativ direkt, weswegen wir auf die Details nicht weiter eingehen.

Unser Fokus liegt auf der Kommunikation der Teilchecker. Die Kommunikation verhält sich in dem vorliegenden Multi-Agenten-System anders als in einem Netzwerk, da jeder Roboter mit jedem anderen Roboter der Flotte benachbart ist.

Hier beobachten wir, dass die Entscheidung der lokalen Prädikate γ_{ex} und γ_{max} für einen Roboter nur von Informationen abhängen, die unabhängig von seinen Nachbarn sind. Hingegen ist es für die Entscheidung des lokalen Prädikats γ_{agree} für einen Roboter notwendig die Teilausgaben all seine Nachbarn zu kennen.

Das bedeutet im Fall eines Multi-Agenten-Systems, dass ein Teilzeuge sehr groß wird, denn er enthält jede Teilausgabe eines jeden Roboters der Flotte. Darüber hinaus bedeutet es auch, dass für die Konsistenzprüfung des Zeugen ein hoher Aufwand für die Kommunikation anfällt.

In unserer Analyse der Invasivität betrachten wir deswegen nur das lokale Prädikat γ_{agree}. Wir stellen außerdem fest, dass anstelle einer Konsistenzprüfung des berechneten Zeugen, jeder Teilchecker seinen Teilzeugen auch während der Prüfung sammeln kann. Der Kommunikationsaufwand ist gleich.

Die Forderung der Konsistenz ist in diesem speziellen Fall äquivalent zur Forderung des Prädikats Γ_{agree}. Es gibt nur für dieses Prädikat einen überlappenden Zeugen, alle geteilten Informationen werden dabei global im gesamten Netzwerk geteilt werden und das Prädikat fordert bereits Gleichheit. Das Prädikat zur Einigkeit ist deswegen ein im allgemeinen einfacherer Spezialfall der Konsistenz.

11.3.3 Kommunikation der Teilchecker

Wir betrachten deswegen die Kommunikation der Teilchecker für die folgenden drei Aufgaben eines Teilcheckers eines Roboters v gesondert:

Aufgabe (1): Einsammeln der Gewinner-Tripel der Nachbarn für den Teilzeugen von v, sowie die Entscheidung des lokalen Prädikats γ_{agree}.

Aufgabe (2): Teilnahme an der Entscheidung der Verteilungsprädikate Γ_{agree}, Γ_{exist} und Γ_{max}.

Aufgabe (3): Teilnahme an der Evaluation des verteilbaren Zeugenprädikats.

11.3.3.1 *Kommunikationsstrukturen*

Für die Aufgabe (1) genügt es aufgrund der Transitivität der Gleichheitsrelation über Nachbarschaften, wenn es eine Kommunikationsstruktur in der Flotte gibt, sodass die Nachbarschaftsrelation einen zusammenhängenden Graphen induziert.

Dieser Umstand motiviert uns, drei solche Kommunikationsstrukturen zu vergleichen. Wir betrachten einen vollständigen Graph, der die Nachbarschaftsrelation des ursprünglichen Systems darstellt. Außerdem betrachten wir zwei virtuelle Netzwerke: einen Stern (ein Baum, bei dem ein Roboter die Wurzel ist und jeder andere Roboter ein Kinder der Wurzel) und einen binären Baum (ein Baum, bei dem jeder Roboter zwei Kinder hat, sofern möglich).

11.3.3.2 *Notation des Vergleichs in der Tabelle*

Wir vergleichen die drei Kommunikationsstrukturen folgend hinsichtlich ihrer Invasivität. Die Ergebnisse sind in der Tabelle in Abbildung 11.3 zusammengefasst.

Dabei ist n die Größe der Flotte, also die Anzahl der Roboter. Die Tabelle listet die Metriken zur Invasivität je Kommunikationsstruktur und Aufgabe (1)-(3) der Teilchecker auf. Als Teil der Notation geben wir dabei an, ob die Anzahl der Roboter, die eine bestimmte lokale Laufzeit haben, konstant ist oder linear von der Größe der Flotte abhängt: Zum Beispiel bedeutet $\Theta(n)_1$, dass eine konstante Anzahl von Robotern die lokale Laufzeit $\Theta(n)$ hat und $\Theta(n)_n$, dass die Anzahl der Roboter, die die lokale Laufzeit $\Theta(n)$ haben, linear von der Größe der Flotte abhängt.

Wir geben außerdem explizit mit 0 statt mit $\Theta(1)$ an, wenn einige Roboter gar keine lokale Berechnung ausführen, um aufzuzeigen, wie gleichverteilt der Berechnungsaufwand auf die Roboter ist.

Weiterhin ist die Zeile zur lokalen Laufzeit für die virtuellen Netzwerke zweigeteilt; in der ersten Zeile notieren wir die lokale Laufzeit der Wurzel mit der Ausnahme der Aufgabe (2) beim binären Baum, wo wir die lokale Laufzeit aller inneren Knoten festhalten. In der zweiten Zeile betrachten wir dann jeweils die verbliebenen Roboter.

	Vollständiger Graph			Stern			Binärer Baum		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
Lokale Laufzeit	$\Theta(n)_n$	$\Theta(n)_n$	$\Theta(1)_n$	$\mathcal{O}_1$	$\Theta(n)_1$	$\Theta(1)_1$	$\mathcal{O}_1$	$\Theta(1)_n$	$\Theta(1)_1$
	-	-	-	$\Theta(1)_n$	$\mathcal{O}_n$	$\mathcal{O}_n$	$\Theta(1)_n$	$\mathcal{O}_n$	$\mathcal{O}_n$
Nachrichtenkomplexität	$\Theta(n^2)$	$\Theta(n^2)$	-	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Laufzeit	$\Theta(1)$	$\Theta(1)$	-	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Abbildung 11.3: Die Tabelle zeigt den Vergleich hinsichtlich der Metriken für die Invasivität je Aufgabe (1)-(3) zur Entscheidung des verteilbaren Zeugenprädikats für das ursprüngliche Multi-Agenten-System; sowie für die virtuellen Netzwerke, Stern und Binärer Baum.

11.3.3.3 *Vollständiger Graph*

Für die Aufgabe (1) sendet jeder Teilchecker das Gewinner-Tripel seines Roboters als Broadcast-Nachricht. Weiterhin entscheidet er das lokale Prädikat γ_{agree}, indem er die empfangenen Tripel mit seinem vergleicht. Folglich ist die lokale Laufzeit linear in der Größe der Flotte für alle Roboter.

Für die Aufgabe (2) sendet jeder Teilchecker die Wahrheitswerte der ausgewerteten lokalen Prädikate als Broadcast-Nachricht. Weiterhin entscheidet er die Verteilungsprädikate mithilfe der empfangenen Wahrheitswerte.

Für die Aufgabe (3) wertet jeder Teilchecker das verteilbare Zeugenprädikat aus, indem er den Wahrheitswert der Konjunktion der Verteilungsprädikate bestimmt.

11.3.3.4 *Stern*

Für die Aufgabe (1) sendet die Wurzel ihr Gewinner-Tripel an all ihre Kinder, also alle anderen Roboter. Weiterhin entscheidet jedes Kind das lokale Prädikat γ_{agree}, indem es das empfangene Tripel mit seinem vergleicht.

Für die Aufgabe (2) sendet jedes Kind die Wahrheitswerte der ausgewerteten lokalen Prädikate an die Wurzel. Weiterhin entscheidet die Wurzel die Verteilungsprädikate mithilfe der empfangenen Wahrheitswerte. Da die Wurzel die Verteilungsprädikate entscheidet, ist die lokale Laufzeit der Wurzel linear in der Größe der Flotte.

Für die Aufgabe (3) wertet die Wurzel das verteilbare Zeugenprädikat aus, indem sie den Wahrheitswert der Konjunktion der Verteilungsprädikate bestimmt und sendet das Ergebnis als Broadcast-Nachricht.

11.3.3.5 *Binärer Baum*

Für die Aufgabe (1) sendet jeder Knoten, der kein Blatt ist, sein Gewinner-Tripel an seine Kinder. Weiterhin entscheidet jedes Kind das lokale Prädikat γ_{agree}, indem es das empfangene Tripel mit seinem vergleicht.

Für die Aufgabe (2) erhält jede Teilchecker, startend bei den Blättern, die Wahrheitswerte der ausgewerteten lokalen Prädikate seiner Kinder. Er kombiniert die Wahrheitswerte entsprechend mit seinen eigenen und sendet sie zu seinem Elternknoten. Weiterhin entscheidet die Wurzel die Verteilungsprädikate mithilfe der empfangenen Wahrheitswerte. Die Laufzeit entspricht der Tiefe des Baums.

Für die Aufgabe (3) wertet die Wurzel das verteilbare Zeugenprädikat aus, indem er den Wahrheitswert der Konjunktion der Verteilungsprädikate bestimmt und sendet das Ergebnis als Broadcast-Nachricht.

11.3.3.6 *Fazit*

Während ein vollständiger Graph als Kommunikationsstruktur durch das Multi-Agenten-System gegeben ist, muss ein virtuelles Netzwerk erst konstruiert werden. Hierdurch entsteht ein Overhead. Andererseits muss dies nur einmalig geschehen. Für den vollständigen Graph müssen die Roboter keine IDs der anderen Roboter kennen, da jede Nachricht als Broadcast-Nachricht verschickt werden kann. Für den Stern hingegen muss die ID der Wurzel und für den binären Baum jeweils die IDs des Elternknotens und der Kinder bekannt sein.

Der vollständige Graph und der Stern haben die geringste Laufzeit. Allerdings performt der vollständige Graph am schlechtesten, wenn es um die Nachrichtenkomplexität und die lokale Laufzeit geht. Im Fall der lokalen Laufzeit führt jeder einzelne Roboter eine globale Berechnung durch. Der binäre Baum hingegen optimiert die lokale Laufzeit.

Wir schlussfolgern, dass der vollständige Graph nicht sinnvoll ist, um die Kommunikation der Teilchecker zu strukturieren. Die Nutzung eines Sterns oder eines binären Baums haben jedoch beide ihre Rechtfertigung. Tatsächlich zeigen sie einen Trade-off zwischen der Laufzeit und der lokalen Laufzeit auf. Ein Stern ist extrem in seinem Verzweigungsfaktor und optimiert somit die Laufzeit.

Ein Pfad wäre hingegen extrem hinsichtlich der Tiefe mit linearer Laufzeit und konstanter lokaler Laufzeit. Wir haben dennoch einen binären Baum gewählt, da er einerseits sublineare Laufzeit hat und andererseits konstante lokale Laufzeit für jeden Roboter. Der Verzweigungsfaktor sollte folglich den Voraussetzungen des Systems angepasst werden.

Außerhalb der vorliegenden Fallstudie betrachten wir Netzwerke mit beliebiger Topologie. Sollte für ein konkretes Netzwerk jedoch eine Topologie bekannt sein, so kann auch dort ein virtuelles Netzwerk eingesetzt werden, um die Invasivität des Checkers zu reduzieren.

Teil V

FORMALE INSTANZVERIFIKATION

Wir stellen in diesem Teil der Arbeit ein Framework für die formale Instanzverifikation mit zertifizierenden verteilten Algorithmen vor, welches wir mit dem Beweisassistenten COQ entwickelt haben.

In Kapitel 12 erläutern wir die Methode für zertifizierende verteilte Algorithmen. Wir stellen eine erste Fallstudie vor, in der wir die Zeugeneigenschaft für die zertifizierende Variante der verteilten Berechnung der kürzesten Pfade in COQ beweisen. Wir diskutieren anschließend die Auswahl zweier Bibliotheken, mit denen wir ein Netzwerkmodell in COQ modellieren.

In Kapitel 13 stellen wir unser Netzwerkmodell in COQ vor und geben eine Übersicht über das Framework zur Umsetzung der formalen Instanzverifikation.

In Kapitel 14 präsentieren wir eine Implementierung und Verifikation in COQ sowohl für die lokale Konsistenzprüfung zusammenhängender Zeugen, als auch für die allgemeine Konsistenzprüfung beliebiger Zeugen.

In Kapitel 15 illustrieren wir die formale Instanzverifikation in unserem Framework anhand weiterer Fallstudien, der zertifizierenden Leader Election und des zertifizierenden verteilten Bipartitheitstest.

12 METHODIK FÜR ZERTIFIZIERENDE VERTEILTE ALGORITHMEN

In diesem Kapitel erläutern wir unsere Methodik für die formale Instanzverifikation zertifizierender verteilter Algorithmen.

In Abschnitt 12.1 stellen wir die Beweisverpflichtungen vor, die sich aus unserer Methode ergeben. Anschließend zeigen wir in Abschnitt 12.2 auf, wie wir den Beweisassistenten COQ zur Lösung der Beweisverpflichtungen im Allgemeinen einsetzen.

In Abschnitt 12.3 stellen wir eine erste Fallstudie in COQ vor. Wir beweisen die Zeugeneigenschaft der in Abschnitt 9.2 vorgestellten zertifizierenden Variante der verteilten Berechnung kürzester Pfade in COQ. Die Fallstudie entstand im Rahmen einer, von der Autorin der vorliegenden Arbeit betreuten, Diplomarbeit [Ash16].

Für diese erste Fallstudie ist noch keine Kommunikation in COQ nötig. In Abschnitt 12.4 stellen wir ausgewählte Bibliotheken vor, mit denen wir ein Netzwerkmodell in COQ modellieren, das auch Kommunikation modelliert.

12.1 BEWEISVERPFLICHTUNGEN

In Abschnitt 12.1.1 leiten wir die entstehenden Beweisverpflichtungen methodisch ab und in Abschnitt 12.1.2 listen wir alle Beweisverpflichtungen im Detail auf.

12.1.1 Methodische Ableitung der Beweisverpflichtungen

Unser Ziel ist eine Methode für die formale Instanzverifikation zertifizierender verteilter Algorithmen. Wir erinnern uns, dass für die formale Instanzverifikation gilt, dass einem Nutzer ein maschinen-geprüfter Beweis für die Korrektheit seines Eingabe-Ausgabe-Paars zur Laufzeit vorliegt, sofern die Laufzeitverifikation nicht fehlschlägt. Im Falle zertifizierender Algorithmen also, falls der Checker akzeptiert.

In dem Theorem 8.2.1 haben wir die Instanzverifikation aus Sicht eines Nutzers festgehalten. Hieraus lassen sich die Beweisverpflichtungen

ableiten, für die es bei der *formalen* Instanzverifikation einen *maschinen-geprüften* Beweis benötigt.

Sei N ein Netzwerk und A ein zertifizierender verteilter Algorithmus, der ein Problem gegeben durch die Spezifikation (ϕ, ψ) lösen soll. Sei Γ ein globales Prädikat in N und $\mathfrak{B}_\Gamma$ ein maschinen-geprüfter Beweis dafür, dass Γ ein verteilbares Zeugenprädikat in N für (ϕ, ψ) ist. Sei weiterhin C ein verteilter Algorithmus und $\mathfrak{B}_C$ ein maschinen-geprüfter Beweis dafür, dass C eine korrekte Konsistenzprüfung der potenziellen Zeugen implementiert und eine korrekte Entscheidungsprozedur für Γ ist.

Wir nehmen nun an, dass A eine Ausgabe o und einen potenziellen Zeugen w für eine Eingabe i berechnet. Dann folgt aus dem Theorem 8.2.1 der Instanzverifikation folgendes: Wenn C für (i, o, w) akzeptiert, dann gibt es einen maschinen-geprüften Beweis dafür, dass $(i, o) \in \psi$ oder $i \notin \phi$.

12.1.2 Liste der Beweisverpflichtungen

Wir listen die Beweisverpflichtungen (BV) im Einzelnen auf und gehen dabei von der lokalen Konsistenzprüfung für zusammenhängende Zeugen aus:

BV I Für Γ gilt:

- **(i)** Γ hat die Zeugeneigenschaft für (ϕ, ψ).
- **(ii)** Γ hat die Verteilungseigenschaft für (ϕ, ψ) in N.

BV II Das Theorem 7.3.4 zur lokalen Konsistenzprüfung ist korrekt.

BV III Der implementierte Checker C ist korrekt:

- **(i)** Jeder Teilchecker entscheidet die Wohlgeformtheit des Teilzeugen seiner Komponente.
- **(ii)** Gemeinsam entscheiden die Teilchecker den Zusammenhang des Zeugen.
- **(iii)** Jeder Teilchecker entscheidet die Konsistenz für die Teilzeugen in seiner Nachbarschaft.
- **(iv)** Jeder Teilchecker entscheidet die lokalen Prädikate von Γ für seine Komponente.
- **(v)** Gemeinsam evaluieren die Teilchecker Γ.

Die Beweisverpflichtung I schlüsselt auf, was für den maschinen-geprüften Beweis $\mathfrak{B}_\Gamma$ zu tun ist. Nur wenn Γ die Zeugen- und die Verteilungseigenschaft hat, ist Γ ein verteilbares Zeugenprädikat.

Die Beweisverpflichtungen II und III zeigen auf, was für den maschinen-geprüften Beweis $\mathfrak{B}_C$ zu tun ist. Dabei beziehen sich die Beweisverpflichtungen III(i)-(v) auf die verschiedenen Aufgaben, die

ein Checker implementiert. Für den Checker gehen wir von einer lokalen Konsistenzprüfung zusammenhängender Zeugen aus, deren Funktionsweise auf dem Theorem 7.3.4 basiert. Zwar existiert ein Beweis für das Theorem auf Papier, aber die formale Instanzverifikation benötigt einen maschinen-geprüften Beweis. Dadurch entsteht die Beweisverpflichtungen II.

12.1.2.1 *Black-Box-Prinzip der formalen Instanzverifikation*

Wir weisen an dieser Stelle darauf hin, dass sich keine der Beweisverpflichtungen auf den verteilten Algorithmus A, also die Konstruktion, bezieht. Konstruktion und Prüfung sind hier von einander getrennt. Für einen Nutzer ist kein Wissen über die Konstruktion nötig.

Wir setzen damit das Black-Box-Prinzip zertifizierender Algorithmen auch bei der formalen Instanzverifikation um.

12.2 EINSATZ DES BEWEISASSISTENTEN COQ

Unser Ziel ist, in diesem Abschnitt, einen Eindruck für das Vorgehen mit dem Beweisassistenten COQ zu vermitteln. Zur Unterstützung der Beweisführung stellt COQ eingebaute *Taktiken* bereit – kleine Programme, die einen Kalkül des natürlichen Schließens umsetzen, aber auch Teile einer Beweissuche automatisieren. Darüber hinaus können auch benutzerdefinierte Taktiken für COQ entwickelt werden [Deloo].

Der Beweis fängt mit dem Beweisziel an und wird durch Taktiken verändert, die neue Ziele generieren, unter deren Voraussetzung das ursprüngliche Ziel bewiesen ist. Es können auch keine Ziele generiert werden. Der Beweis ist dann abgeschlossen, wenn es keine Beweisziele mehr gibt.

Einige Taktiken stellen wir im Folgenden vor. Für weitere Taktiken verweisen wir auf die COQ -Webseite, die einen Index mit Erklärungen der Taktiken bereitstellt [Coqb].

In Abschnitt 12.2.1 stellen wir einfache Taktiken vor, die wir in Abschnitt 12.2.2 an einem kleinen Beispiel demonstrieren. Wie wir Taktiken kombinieren können, zeigen wir in Abschnitt 12.2.3. In Abschnitt 12.2.4 führen wir komplexere Taktiken und die Nutzung von Bibliotheken ein. In Abschnitt 12.2.5 diskutieren wir die Programmierung in COQ. In Abschnitt 12.2.6 stellen wir schließlich die Beweisprüfung vor, die in COQ äquivalent zur Typprüfung von Programmen ist.

12.2.1 Einfache Taktiken

COQ setzt einen Kalkül des natürlichen Schließens um [PM15]. Die Einführungs- und Beseitigungsregeln der logischen Junktoren sind in den Tabellen 12.1 und 12.2 dargestellt.[1] Die Tabellen enthalten die Regeln für den Wahrheitswert „Falsch“ ($\bot$), die Negation ($\neg$), die Implikation ($\to$), den Allquantor ($\forall$), die Konjunktion ($\wedge$), die Disjunktion ($\vee$), den Existenzquantor ($\exists$) und die Gleichheit ($=$).

In beiden Tabellen steht in der linken Spalte das mathematische Symbol, in der mittleren die Einführungs- oder Beseitigungsregeln und in der rechten Spalte die COQ-Taktik, deren Argumente kursiv dargestellt sind. Zwei Terme in COQ sind gleich (Notation: $\equiv$), wenn sie nach der Auswertung (in dem implementierten λ-Kalkül) den gleichen Wert haben.

Der Beweiskontext Γ ist eine Liste mit Objekten der Form $x : T$, wobei x ein Name und T ein Term eines bestimmten Typs ist. Für diese Arbeit sind vor allem Terme T vom Typ „logische Aussage“ relevant. Die Notation $\Gamma \vdash x : T$ bedeutet für logische Aussagen, dass T in Γ beweisbar ist und x ein Beweis dafür ist [PM12]. Der Beweiskontext enthält, vereinfacht gesagt, alle Annahmen und Variablen.

Symbol	Einführungsregel	Taktik
$\bot$		
$\neg$	$\dfrac{\Gamma, h : A \vdash ? : \bot}{\Gamma \vdash ? : \neg A}$	**intro *h***
$\to$	$\dfrac{\Gamma, h : A \vdash ? : B}{\Gamma \vdash ? : A \to B}$	**intro *h***
$\forall$	$\dfrac{\Gamma, y : A \vdash ? : B[x \leftarrow y]}{\Gamma \vdash ? : \forall x : A, B}$	**intro *y***
$\wedge$	$\dfrac{\Gamma \vdash ? : A \quad \Gamma \vdash ? : B}{\Gamma \vdash ? : A \wedge B}$	**split**
$\vee$	$\dfrac{\Gamma \vdash ? : A}{\Gamma \vdash ? : A \vee B}$ $\dfrac{\Gamma \vdash ? : B}{\Gamma \vdash ? : A \vee B}$	**left** **right**
$\exists$	$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash ? : B[x \leftarrow t]}{\Gamma \vdash ? : \exists x : A, B}$	**exists *t***
$=$	$\dfrac{t \equiv u}{\Gamma \vdash ? : t = u}$	**reflexivity**

Tabelle 12.1: Logische Einführungsregeln mit jeweils der entsprechenden Taktik.

1 Vergleiche mit Paulin-Mohrings Tabelle [PM12, S.7]. Wir haben die Tabelle für uns angepasst und die Notation übernommen.

Symbol	Beseitigungsregel	Taktik
$\bot$	$\dfrac{\Gamma \vdash ? : \bot}{\Gamma \vdash ? : C}$	**exfalso**
$\neg$	$\dfrac{\Gamma \vdash h : \neg A \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : C}$	**destruct *h***
$\rightarrow$	$\dfrac{\Gamma \vdash h : A \rightarrow B \quad \Gamma \vdash ? : A}{\Gamma \vdash ? : B}$	**apply *h***
$\forall$	$\dfrac{\Gamma \vdash h : \forall x : A, B \quad \Gamma \vdash t : A}{\Gamma \vdash ? : B[x \leftarrow t]}$	**apply *h* with (*x*:=*t*)**
$\wedge$	$\dfrac{\Gamma \vdash h : A \wedge B \quad \Gamma, l : A, m : B \vdash ? : C}{\Gamma \vdash ? : C}$	**destruct *h* as [*l* *m*]**
$\vee$	$\dfrac{\Gamma \vdash h : A \vee B \quad \Gamma, l : A \vdash ? : C \quad \Gamma, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	**destruct *h* as [*l*|*l*]**
$\exists$	$\dfrac{\Gamma \vdash h : \exists x : A, B \quad \Gamma, x : A, l : B \vdash ? : C}{\Gamma \vdash ? : C}$	**destruct *h* as [*x* *l*]**
$=$	$\dfrac{\Gamma \vdash h : t = u \quad \Gamma \vdash ? : C}{\Gamma \vdash ? : C[x \leftarrow t]}$	**rewrite *h***

Tabelle 12.2: Logische Beseitigungsregeln mit jeweils der entsprechenden Taktik.

In den Tabellen steht bei den Regeln an einigen Stellen ein Fragezeichen anstelle eines Namens für eine Aussage. Das bedeutet, dass es für diese Aussage keinen Beweis geben muss, um die Taktik der Regel anwenden zu können. Wenn die Voraussetzung einer Regel ein Fragezeichen enthält, dann wird bei der Anwendung der entsprechenden Taktik ein neues Beweisziel für diese Aussage generiert. Die Regeln sind in COQ „rückwärts“ implementiert. Das heißt, dass das zu manipulierende Beweisziel die Schlussfolgerung der Regel ist. In der Schlussfolgerung einer Regel steht deswegen immer ein Fragezeichen.

Eine Einführungsregel wird in COQ genutzt, wenn das Beweisziel den entsprechenden Junktor enthält. Wenn das Beweisziel zum Beispiel eine Implikation $A \rightarrow B$ ist, dann führt die `intro`-Taktik als neues Ziel B ein und erweitert den Beweiskontext um die Annahme A. Eine Beseitigungsregel wird in COQ genutzt, wenn es eine Annahme mit dem Junktor gibt. Zum Beispiel wird mit der `apply`-Taktik der Modus Ponens umgesetzt.

Eine Taktik für den Fall, dass das Beweisziel einer Annahme h entspricht, ist `exact`:

$$\frac{h : A \in \Gamma}{\Gamma \vdash h : A}$$

Die `assumption`-Taktik ist eine allgemeinere Form von `exact`, denn sie sucht selbst eine passende Annahme im Beweiskontext. Mit diesen Taktiken wird zum Beispiel kein neues Beweisziel generiert.

12.2.2 Einfache Taktiken anhand eines Beispiels

Am Beispiel der Kommutativität der Konjunktion zeigen wir, wie ein Beweis mit einfachen Taktiken aussieht. [2] Der Code-Block 12.1 zeigt das Lemma und einen Beweis. Mit dem Schlüsselwort `Lemma` führen wir das Beweisziel mit einem Namen, hier `commutativity_and`, ein. Die Variablen haben den Typ `Prop`, was in COQ der Typ für logische Aussagen ist. Das Schlüsselwort `Proof` startet den interaktiven Beweismodus und wird nach Abschluss des Beweises mit dem Schlüsselwort `Qed` beendet. COQ prüft dann final die Korrektheit des geführten Beweises.

```
Lemma commutativity_and:
 forall A B: Prop, A ∧ B → B ∧ A.
Proof.
 intro A. intro B. intro H.
 split.
 destruct H as [a b].
 exact b.
 destruct H as [a b].
 exact a.
Qed.
```

Code-Block 12.1: Lemma und Beweis der Kommutativität der Konjunktion mit einfachen Taktiken.

Beweisskripte, die nicht interaktiv in COQ ausgeführt werden, sind im Allgemeinen schwierig zu verstehen. Der folgende Beweisbaum soll deshalb die einzelnen Schritte des Skripts aus Code-Block 12.1 verdeutlichen:

$$\dfrac{\dfrac{\dfrac{b : B \in \Gamma, A : \mathrm{Prop}, B : \mathrm{Prop}, a : A, b : B}{\Gamma, A : \mathrm{Prop}, B : \mathrm{Prop}, a : A, b : B \vdash ? : B}\ \texttt{exact b} \qquad \dfrac{\vdots}{\ldots \vdash ? : A}}{\Gamma, A : \mathrm{Prop}, B : \mathrm{Prop}, H : A \wedge B \vdash ? : B}\ \texttt{destruct H}}{\Gamma, A : \mathrm{Prop}, B : \mathrm{Prop}, H : A \wedge B \vdash ? : B \wedge A}\ \texttt{split}}{\dfrac{\dfrac{\Gamma, A : \mathrm{Prop}, B : \mathrm{Prop} \vdash ? : A \wedge B \to B \wedge A}{\Gamma, A : \mathrm{Prop} \vdash ? : \forall B : \mathrm{Prop}, A \wedge B \to B \wedge A}\ \texttt{intro B}}{\Gamma \vdash ? : \forall A\ B : \mathrm{Prop}, A \wedge B \to B \wedge A}\ \texttt{intro A}}\ \texttt{intro H}$$

Die Wurzel des Beweisbaumes ist hier unten dargestellt und sie ist das Beweisziel. Die Kindsknoten im Baum sind jeweils die Voraussetzungen der jeweiligen Regel. Die Blätter haben keine Voraussetzung mehr, deswegen wird auch kein neues Ziel generiert. Der Beweisbaum verzweigt durch die Einführungsregel der Konjunktion, die als einzige verwendete Regel mehr als eine Voraussetzung hat. Es ist nur die Verzweigung für das Teilziel $? : B$ dargestellt, der andere Zweig ist nur angedeutet und verläuft analog.

2 Wir verwenden ein Beispiel aus [Vö13].

Für jede benutzte Regel steht die Taktik, die diese implementiert am Baum. Wenn der Baum von der Wurzel aus durchlaufen wird, dann ergeben sich die Taktiken in der Reihenfolge des Skripts. Die `destruct`-Taktik steht im Baum, aus Platzgründen, ohne ein Zerlegungsschema. Im Skript sehen wir jedoch, dass die Annahme H in die beiden Teile der Konjunktion, benannt a und b, zerlegt wird. Der Beweis wird abgeschlossen, weil b : B (und im anderen Zweig a : A) als Annahme im Beweiskontexts enthalten ist.

Der Beweis für dieses Beispiel erscheint bereits verhältnismäßig aufwendig und so schreibt Lescuyer über Beweisassistenten [Les11, S.5-6]:

> *Unfortunately, they can be very tedious to work with because proofs must be justified by small basic steps and therefore require much more detail than even the most detailed pencil-and-paper proof.*

Die Belohnung der aufwendigen Beweisführung ist die große Gewissheit über die Korrektheit des geführten Beweises.

Da wir bisher nur einfache Taktiken betrachtet haben, kann der Beweis in diesem Fall auch noch gekürzt werden, wie wir im folgenden Abschnitt zeigen. Dennoch sind Beweise, die wir in einem Beweisassistenten führen, im Allgemeinen exakter und detaillierter als solche, die wir auf Papier führen.

12.2.3 Taktiken kombinieren

Wir können in COQ auch Taktiken kombinieren [Deloo]. Hier sind einige Beispiele dafür:

`t1;t2` Die Taktik `t1` wird auf das aktuelle Ziel und anschließend die Taktik `t2` auf alle resultierenden Ziele angewandt.

`t;[t1|..|tn]` Erst wird die Taktik `t` ausgeführt, die `n` Ziele generiert. Auf dem `i`-ten Teilziel wird dann die Taktik `ti` angewandt.

`repeat t` Die Taktik `t` wird rekursiv angewandt bis sie fehlschlägt.

`first [t1|...|tn]` Die erste Taktik, die keinen Fehler produziert, wird benutzt.

`solve [t1|...|tn]` Die erste Taktik, die keinen Fehler produziert und keine weiteren Ziele generiert, wird benutzt.

Den Beweis für das Lemma zur Kommutativität können wir mithilfe dieser Kombinationen vereinfachen. In der Abbildung 12.2 sehen wir den so vereinfachten Beweis. Mit `repeat` wird die `intro`-Taktik mehrfach aufgerufen – dafür hat COQ eine eigene Taktik namens `intros`. Die `split`-Taktik generiert zwei Teilziele, auf beide werden die folgenden Taktiken gleichermaßen angewandt, da die Taktiken mit `;` kombiniert werden.

```
Lemma commutativity_and:
 forall A B: Prop, A ∧ B → B ∧ A.
Proof.
 repeat intro.
 split ; destruct H as [a b] ; [exact b | exact a].
Qed.
```

Code-Block 12.2: Lemma und Beweis der Kommutativität der Konjunktion mit kombinierten Taktiken.

12.2.4 Komplexere Taktiken und Bibliotheken

COQ verfügt über komplexere Taktiken, die elementare Schritte kombinieren oder eine Beweissuche implementieren und so für einen höheren Automatisierungsgrad sorgen.

Die `tauto`-Taktik implementiert eine Entscheidungsprozedur für die intuitionistische Logik [Dyc92]. Sie findet zum Beispiel automatisch einen Beweis für die Kommutativität der Konjunktion.

Die `intuition`-Taktik benutzt Informationen des Suchbaums, den die `tauto`-Taktik aufgebaut, um Beweisziele zu generieren, die äquivalent zum zu lösenden Beweisziel sind. Für diese Beweisziele wird dann mit der `auto`-Taktik ein Beweis gesucht, unter anderem in den Lemmata der COQ-Standardbibliothek durchsucht werden.

Die Standardbibliothek besitzt verschiedene Module, zum Beispiel zur klassischen Logik, Strukturen, Arithmetik, natürlichen Zahlen, reellen Zahlen, Vektoren, Mengen oder Listen. Weitere Bibliotheken können zusätzlich eingebunden werden. Für unser Framework benutzen wir zusätzlich eine Bibliothek für Graphen und eine für verteilte Systeme.

12.2.5 Programmieren in COQ

COQ hat eine funktionale Programmiersprache: Gallina. Die in Gallina geschriebenen Programme können direkt in COQ ausgeführt werden. Die Standardbibliothek von COQ bietet nur einen recht überschaubaren Satz von in Gallina vordefinierten Datentypen und Funktionen. Gallina stellt jedoch mächtige Mechanismen bereit, um eigene Datentypen zu definieren und Funktionen zu programmieren.

12.2.5.1 *Funktionen*

Übliche Mechanismen zum Programmieren von Funktionen in funktionalen Programmiersprachen sind Pattern Matching über die Datentypen der Argumente einer Funktion oder eine rekursive Definitionen.

Ein Beispiel für zwei Funktionen in COQ ist in Code-Block 12.3 zusehen. Die Funktion `is_empty` gibt für eine Liste `l` von natürlichen Zahlen `true` zurück, falls `l` leer ist und ansonsten `false`. Implementiert ist `is_empty` als Pattern Matching über die Konstruktoren des Datentyps Liste, hier `nil` und `element :: tail`.

Die Funktion `plus` ist eine rekursive Funktion, die zwei natürliche Zahl addiert. Ein Pattern Matching über die Konstruktoren der natürlichen Zahlen unterscheidet den Basisfall vom Rekursionsfall.

```
Definition is_empty ( l : list nat ) : bool :=
    match l with
    | nil => true
    | element :: tail => false
    end.

Fixpoint plus ( n : nat ) ( m : nat ) : nat :=
    match n with
    | O => m
    | S 'n => plus 'n ( S m )
    end.
```

Code-Block 12.3: Beispiel für Funktionen in COQ.

COQ hat ein ausdrucksstarkes Typsystem. Wir verzichten an dieser Stelle jedoch auf eine weitere Einführung. Wir vermuten, dass die meisten Leser:innen zumindest mit einer funktionalen Programmiersprache bereits vertraut sind. Das sollte ausreichen, um den Programmen für diese Arbeit zumindest intuitiv folgen zu können. Aspekte der funktionalen Programmierung, die spezifisch für COQ sind, erläutern wir an den entsprechenden Stellen.

Für eine Vertiefung in die Programmierung in COQ empfehlen wir den interessierten Leser:innen ansonsten Chlipalas Buch [Chl13] über funktionale Programmierung in COQ.

12.2.5.2 *Programmextraktion*

Eine Besonderheit in COQ ist die Programmextraktion. Funktionale Programme können durch einen implementierten Mechanismus der *Extraktion* in einige funktionale Sprachen übersetzt werden. Das ist interessant, weil COQs funktionale Sprache GALLINA nicht Turingmächtig ist und COQ keinen Compiler bereitstellt, um ausführbaren Code zu erzeugen. Die Extraktion gelingt per „Knopfdruck“ von einem Programm in GALLINA in die funktionalen Programmiersprachen SCHEME, OCAML oder HASKELL [Leto8]. Das heißt für die Extraktion fällt in COQ keine zusätzliche Arbeit an.

Dabei ist insbesondere zu erwähnen, dass bei der Extraktion alle für das GALLINA-Programm bewiesenen Eigenschaften auch in den Ziel-

sprachen erhalten bleiben, sofern der Mechanismus zur Extraktion korrekt ist. Bei der Extraktion wird von Gallina-Quellcode beispielsweise nach OCaml-Quellcode übersetzt und dann mit dem OCaml-Compiler ein ausführbares Programm erzeugt. Folglich muss der OCaml-Compiler fehlerfrei sein, damit die Kompilierung gelingt. In [Mul+18] wird deswegen ein verifizierter Compiler vorgestellt mit dem Gallina-Quellcode zu einem ausführbaren Programm übersetzt wird.

12.2.6 Beweisprüfung in COQ

Die funktionale Programmiersprache Gallina ist aufgrund des Typsystems (dependent types) nicht Turing-mächtig. Das Typsystem ist in COQ so gewählt, dass es den Curry-Howard-Isomorphismus zwischen einem typisierten Lambda-Kalkül und einer konstruktiven Logik ermöglicht [Wad15]. Kurz zusammengefasst gilt durch den Isomorphismus: Aussagen sind Typen, Beweise sind Programme. Ein Beweis über Aussagen verhält sich wie ein Programm mit dem entsprechenden Typ. Als Folge ist die Beweisprüfung dann auch Typprüfung. In COQ wird mit dem Befehl `Qed` nicht nur der interaktive Beweismodus beendet, sondern auch die Beweisprüfung gestartet.

12.3 FALLSTUDIE: ZEUGENEIGENSCHAFT FÜR KÜRZESTE PFADE

Wir stellen in diesem Abschnitt eine erste Fallstudie zur formalen Instanzverifikation in COQ vor. Dafür beziehen wir uns auf die vorgestellte zertifizierende Variante der verteilten Berechnung kürzester Pfade (siehe Abschnitt 9.2). Die Fallstudie entstand im Rahmen Arbeit [Ash16] – eine Diplomarbeit, die die Autorin dieser Arbeit betreute.

In Abschnitt 12.3.1 geben wir einen kurzen Überblick über die Fallstudie und führen die nötigen Datentypen ein. In Abschnitt 12.3.2 stellen wir ausgewählte Ausschnitte aus der Fallstudie vor und in Abschnitt 12.3.3 diskutieren wir diese Fallstudie.

12.3.1 Überblick

Wir beweisen in COQ, dass das Zeugenprädikat der zertifizierenden verteilten Berechnung kürzester Pfade die Zeugeneigenschaft, sowie die Verteilungseigenschaft besitzt. In der Liste unserer Beweisverpflichtungen (siehe Abschnitt 12.1.2) für die formale Instanzverifikation

handelt es sich dabei um die Beweisverpflichtung I mit ihren beiden Teilverpflichtungen (i) und (ii).

Wir konzentrieren uns im Rahmen dieser Arbeit nur auf die Zeugeneigenschaft, da die Verteilungseigenschaft der gewählten Fallstudie verhältnismäßig einfach zu zeigen ist (siehe Abschnitt 9.2).

12.3.1.1 *Auswahl der Fallstudie*

Die Fallstudie der kürzesten Pfade hat in der Welt der Programmverifikation eine weitreichende Tradition [Cha11; BLW08; Riz14]. Laut Rizkallah [Riz14, S. 50] gibt es eine „endlose Faszination“ für diese Fallstudie zur Demonstration einer Verifikationsmethode. Für unsere Zwecke ist die Fallstudie deswegen eine sinnvolle Wahl.

12.3.1.2 *Modellierung der Topologie in* COQ

Wir benötigen für die Fallstudie keine Modellierung einer Kommunikation der Komponenten, da wir für diese erste Fallstudie noch keinen verteilten Algorithmus in COQ implementieren. Dennoch benötigen wir eine Modellierung der Topologie eines Netzwerks.

Dafür definieren wir uns in COQ Graphen als Datentyp. Der Code-Block 12.4 zeigt die gewählte Definition mithilfe eines `Record`.

```
Record graph : Set := mk_graph {
    V : nat;
    E : (set V) → (set V) → nat;
    s : (set V)
}.
```

Code-Block 12.4: Definition eines Graphen als `Record` in COQ.

Die Menge der Knoten `V` ist durch deren Anzahl definiert. Die Menge der Kanten inklusive ihrer Kosten ist mit `E` definiert. Dabei gilt, dass jeder vorhandenen Kanten Kosten von mindestens `1` zugeordnet werden. Da wir eine Kante mit Kosten `0` als nicht zum Graphen gehörend interpretieren. Für die Berechnung der kürzesten Pfade benötigen wir eine Quelle; diese wird als besonderer Knoten `s` definiert.

12.3.1.3 *Pfade*

Da wir über kürzeste Pfade in einem Graphen argumentieren wollen, müssen wir auch Pfade in COQ definieren. Der Code-Block 12.5 zeigt die Definition für Pfade. Die Definition eines Pfades `path` benötigt als Argument den Graphen `g`, in dem der Pfad definiert ist. Zur Vereinfachung der Beweisführung, führen wir weitere Informationen mit, wie zum Beispiel alle Knoten des Pfades (`set g.(V)`).

Ansonsten besteht der induktive Datentyp `path` aus den beiden Konstruktoren `zerop` und `consp`. Mit dem Konstruktor `zerop` wird ein leerer Pfad von einem Knoten zu sich selbst erzeugt mit Kosten `0`. Der Konstruktor `consp` erweitert den Pfad um eine Kante und erhöht die Summe der Kosten entsprechend.

```
Inductive path (g : graph) :
    list (set g.(V)) → set g.(V) → set g.(V) → nat → Prop :=
    | zerop : forall v, path g (v:: nil) v v 0
    | consp : forall u v, g.(E) u v > 0 →
        forall p sv d, path g (u:: p) u sv d →
        path g (v:: u:: p) v sv (d + g.(E) u v).
```

Code-Block 12.5: Definition eines Pfads als induktiver Datentyp in COQ.

12.3.2 Zeugeneigenschaft in COQ

Für die Zeugeneigenschaft müssen wir zeigen, dass eine berechnete Funktion der Distanzfunktion entspricht, falls sie die drei Eigenschaften der Charakterisierung erfüllt – für die Definition der Distanzfunktion und deren Charakterisierung siehe Abschnitt 9.2.

12.3.2.1 *Charakterisierung der Distanzfunktion in* COQ

Dafür definieren wir die Distanzfunktion zunächst in COQ, wie in Code-Block 12.6 dargestellt. Dabei ist `dist` eine Variable vom Typ der Distanzfunktion.

```
Variable dist : set g.(V) → nat.
Definition source_prop :=
    dist g.(s) = 0.
Definition triangle_prop := forall u v,
    g.(E) u v > 0 → dist v <= dist u + g.(E) u v.
Definition justification_prop := forall v,
    v <> g.(s) → exists u, g.(E) u v > 0 ∧ dist v = dist u + g.(E) u v.
```

Code-Block 12.6: Charakterisierung der Distanzfunktion in COQ.

Dabei entsprechen die Definitionen genau den drei Eigenschaften der Charakterisierung: Die Definition `source_porp` entspricht der Eigenschaft (9.1), die Definition `triangle_prop` der Eigenschaft (9.2) und die Definition `justification_prop` der Eigenschaft (9.3).

12.3.2.2 *Beweis der Zeugeneigenschaft*

Der Code-Block 12.7 zeigt das Theorem der Zeugeneigenschaft mit entsprechendem Beweis in COQ. Mit `Hypothesis` nehmen wir an,

dass die drei Eigenschaften für die berechnete Distanz `dist` in dem gesamten `COQ`-Modul zur Zeugeneigenschaft gelten.

Dabei ist die Funktion `dist` eine beliebige Funktion, welche die Charakterisierung erfüllt. Sie repräsentiert die berechnete Distanz. Die Funktion `delta` hingegen ist die Distanzfunktion. Wir haben ihre Definition in COQ an dieser Stelle nicht gezeigt. Sie ist analog definiert zu der Definition auf Papier aus Abschnitt 9.2.

Für das Theorem der Zeugeneigenschaft müssen wir folglich zeigen, dass die berechnete Distanz `dist` der korrekten Distanz `delta` entspricht.

```
Hypothesis Hsource_prop : source_prop.
Hypothesis Htrian_prop : triangle_prop.
Hypothesis Hjustf_prop : justification_prop.

Theorem witness_prop : forall v,
    dist v = delta v.
Proof.
    intro v.
    apply le_antisym.
    apply dist_leq_delta.
    apply dist_geq_delta.
Qed.
```

Code-Block 12.7: Zeugeneigenschaft mit Beweis in COQ.

Die Lemmata `dist_leq_delta` und `dist_geq_delta` sind dabei von uns eingeführte Hilfslemmata. Mit Erstem beweisen wir, dass die berechnete Distanz kleiner oder gleich der Distanz pro Knoten ist und mit Zweitem, dass sie größer oder gleich ist. Das Lemma `le_antisym` hingegen stammt aus der Standardbibliothek von COQ und beschreibt die Antisymmetrie der Kleiner-Gleich-Relation.

12.3.2.3 *Hilfslemmata*

Der Beweis der Zeugeneigenschaft benötigt in unserer Fallstudie insgesamt acht Hilfslemmata. Wobei sich dabei einige Lemmata der Eigenschaften der Distanzfunktion widmen. So zeigen wir beispielsweise, dass das Problem der kürzesten Pfade eine optimale Substruktur [Bae19] hat. Darüber hinaus gibt es Lemmata zu technischen Vereinfachungen bei der Beweisführung mit den eingeführten Datentypen `graph` und `path`.

Zusätzlich zu den Lemmata kommen auch Axiome aus der Standardbibliothek zum Einsatz, wie zum Beispiel das Auswahlaxiom [ML06].

12.3.2.4 *Verteilungseigenschaft*

Für die Verteilungseigenschaft benötigen wir entsprechende lokale Prädikate je Komponente eines Netzwerks. Zu zeigen ist, dass falls die lokalen Prädikate erfüllt sind, dann liegt die Funktion `dist` im Netzwerk mit den drei Eigenschaften vor. Der Beweis in COQ kann in [Ash16] eingesehen werden.

12.3.3 Diskussion

Unsere erste Fallstudie in COQ zur formalen Instanzverifikation zeigt uns, dass eine Umsetzung der Beweisverpflichtung I in COQ möglich ist. Die Definitionen der Datentypen Graph und Pfad sind dabei jedoch noch speziell auf die Fallstudie zugeschnitten.

Bisher haben wir auch noch keine Programmverifikation eines Checkers in COQ gesehen. Dafür müssen wir den Checker zunächst als verteilten Algorithmus implementieren und benötigen entsprechend ein Modell eines Netzwerks in COQ (siehe Kapitel 13).

12.4 AUSWAHL DER BIBLIOTHEKEN: TOPOLOGIE UND KOMMUNIKATION IN COQ

Für unser Netzwerkmodell brauchen wir zum einen eine Modellierung der Topologie und zum anderen eine Modellierung der Kommunikation. Wir integrieren zwei Bibliotheken für den Beweisassistenten COQ, um ein Netzwerkmodell als Grundlage für das Framework aufzubauen.

Für die Topologie bieten sich Graphen an. Wir geben deswegen in Abschnitt 12.4.1 einen kurzen Überblick über Graphen in COQ und diskutieren dann unsere Wahl der Bibliothek GraphBasics in Abschnitt 12.4.2. Wir führen außerdem die Konstrukte aus GraphBasics ein, die wir für unser Netzwerkmodell benötigen.

Für die Kommunikation gehen wir analog vor. In Abschnitt 12.4.3 geben wir einen kurzen Überblick zu Arbeiten, die sich mit Kommunikation in COQ beschäftigen. In Abschnitt 12.4.4 diskutieren wir die von uns gewählte Bibliothek Verdi in Hinblick auf unser Netzwerkmodell.

12.4.1 Graphen in COQ

Gewisse Berühmtheit erreichte der Beweis des 4-Farben-Theorems, der erstmalig in COQ geführt wurde [Gon08]. Das Besondere an diesem

Beweis ist, dass er mithilfe eines Programms in COQ geführt wurde, welches eine große Anzahl von Fällen unterscheidet. Die Repräsentation von Graphen in [Gono8] ist auf die kombinatorischen Methoden des geführten Beweises ausgelegt und deswegen für unsere Zwecke nicht allgemein genug.

Wir finden in der Literatur weitere Beweise zu Graphen in COQ, bei denen jedoch die Repräsentation der Graphen auch problemspezifisch gewählt ist [Pot15; Che+18]. Uns sind für COQ die folgenden drei Bibliotheken, die Graphen beinhalten, bekannt: CoLoR, ssreflect und GraphBasics.

CoLoR-Bibliothek. Die Bibliothek CoLoR ist für Graphersetzungssysteme ausgelegt [BK12]. Es geht darum Beweise zu Terminierungseigenschaften zu führen. Die Bibliothek eignet sich nicht für unsere Zwecke.

ssreflect-Bibliothek. Die Bibliothek ssreflect beinhaltet eine Repräsentation von Graphen [WAG12]. Allerdings bringt ssreflect umfangreiche Erweiterungen der funktionalen Programmiersprache Gallina, sowie der Taktiksprache LTac mit sich. Um mit den definierten Datentypen arbeiten zu können, muss man sich in diese umfangreichen Erweiterungen einarbeiten. Dieser Aufwand übersteigt für uns den Nutzen.

GraphBasics-Bibliothek. Die Bibliothek GraphBasics stellt Definition für einige Grundkonzepte der Graphentheorie bereit, wie Pfade oder Bäume [Dupo1]. Die Bibliothek ist verständlich angelegt, für unsere Zwecke gut genug dokumentiert und somit ohne viel Einarbeitungszeit für uns nutzbar.

12.4.2 Nutzung der Bibliothek `GraphBasics`

Wir stellen folgend die Konzepte der Bibliothek GraphBasics vor, die für unser Netzwerkmodell relevant sind.

12.4.2.1 *Knoten und Kanten*

In Code-Block 12.8 ist die Definition für Knoten und Kanten (für Graphen) dargestellt. Ein Knoten `Vertex` ist ein induktiv definierter Typ mit einem Konstruktor `index`, der aus einer natürlichen Zahl einen Knoten `Vertex` konstruiert.

```
Inductive Vertex : Set :=
    index : nat → Vertex .
```

Code-Block 12.8: Definition von Knoten in GraphBasics.

In Code-Block 12.9 ist die Definition gerichteter und ungerichteter Kanten gegeben. Die Menge `A_set` ist dabei die Menge aller gerichteten Kanten und `E_set` erstellt aus einer gerichteten Kante zwei gerichtete Kanten.

```
Inductive Arc : Set :=
    A_ends : Vertex → Vertex → Arc.

Inductive E_set ( x y : Vertex ) : A_set :=
    | E_right : E_set x y ( A_ends x y )
    | E_left : E_set x y ( A_ends y x ).
```

Code-Block 12.9: Definition gerichteter Kanten in GraphBasics.

In GraphBasics benutzen wir außerdem die Notation `v x` für einen Knoten `x`, der in einer Knotenmenge `v` enthalten ist und analog `e y` für eine Kante `y`, die in einer Kantenmenge `e` enthalten ist.

12.4.2.2 *Zusammenhängende Graphen*

Wir wollen die Topologie eines Netzwerks mit einem zusammenhängenden, ungerichteten Graphen modellieren. In Code-Block 12.10 ist die Definition `Connected` für einen solchen Graphen gegeben.

```
Inductive Connected : V_set → A_set → Set :=
    | C_isolated : forall x : Vertex , Connected ( V_single x )
        A_empty
    | C_leaf : forall ( v : V_set ) ( a : A_set )
        ( co : Connected v a ) ( x y : Vertex ),
        v x → not v y →
        Connected ( V_union ( V_single y ) v )
        ( A_union ( E_set x y ) a )
    | C_edge : forall ( v : V_set ) ( a : A_set )
        ( co : Connected v a ) ( x y : Vertex ),
        v x → v y → x <> y →
        not a ( A_ends x y ) → not a ( A_ends y x ) →
        Connected v ( A_union ( E_set x y ) a )
    | C_eq : forall ( v v' : V_set ) ( a a' : A_set ),
        v = v' → a = a' → Connected v a → Connected v' a'.
```

Code-Block 12.10: Definition zusammenhängender Graphen in GraphBasics.

Der Typ Connected hat vier Konstruktoren: `C_isolated` für einzelne Knoten; `C_leaf` für das Hinzufügen eines neuen Knotens `y`; `C_edge` für das Hinzufügen einer neuen Kante zwischen zwei Knoten `x` und `y`, die bereits zum Graphen `co` gehören; `C_eq` dafür, dass zwei Graphen mit unterschiedlicher Benennung der Knoten gleich sind.

12.4.2.3 *Nachbarschaft*

In Code-Block 12.11 ist die Definition der Nachbarschaft eines Knotens dargestellt. Die Nachbarschaft eines Knotens x ist als rekursive Funktion definiert. Sie berechnet für eine Liste gerichteter Kanten la jeweils alle „rechten" Nachbarn – für jede Kante den Knoten an der zweiten Position enthält. Entsprechend gibt es auch eine Funktion, die die linken Nachbarn berechnet.

```
Fixpoint A_in_neighborhood ( x : Vertex ) ( la : A_list )
    { struct la } : V_list :=
    match la with
    | nil => V_nil
    | A_ends x' y' :: la' =>
        if V_eq_dec x y'
        then x' :: A_in_neighborhood x la'
        else A_in_neighborhood x la'
    end.
```

Code-Block 12.11: Definition von Nachbarschaften in GraphBasics.

12.4.3 Kommunikation in COQ

Es gibt einige Arbeiten zu verteilten Algorithmen in COQ [CFM09; ACD16; CM12]. Diese beschäftigen sich jedoch mit der Modellierung verteilter Systeme in COQ und nicht mit der Möglichkeit einen verteilten Algorithmus zu implementieren. Zum Beispiel werden in [CFM09] verteilte Systeme durch Graphersetzungssysteme modelliert.

Unser Ziel ist es in COQ verifizierte Checker zu implementieren, die dann (per Extraktion) auf realen Netzwerken laufen können. Da GALLINA eine rein funktionale Programmiersprache ist und entsprechend keine Nebeneffekte erlaubt, ist das nicht ohne weiteres möglich.

Für die Kommunikation in einem realen Netzwerk bedarf es das Aufrufen von Betriebssystemfunktionen, die mit Nebeneffekten verbunden sind. In funktionalen Programmiersprachen, wie zum Beispiel Haskell, löst man dieses Problem mit Monaden [Wad95] – eine Monade ist ein abstrakter Datentyp, der vom gleichnamigen Konzept aus der Kategorientheorie abgeleitet ist.

Uns sind für COQ drei Bibliotheken bekannt, die die Kommunikation verteilter Systeme behandeln.

12.4.3.1 Ynot-*Bibliothek*

Mit der Bibliothek YNOT wird GALLINA um Monaden im Stil von HASKELL erweitert [WMM10]. Das Problem bei der Verwendung von YNOT

ist jedoch, dass die Netzwerkkommunikation dabei nicht in COQ verifiziert werden kann. In [JTL12] wird die Implementierung und Verifikation eines Web-Browsers in COQ mit YNOT beschrieben. Damit der extrahierte Browser reaktiv mit dem Betriebssystem interagieren kann, ist ein zusätzliches Programm nötig. Die Autoren verifizieren dieses außerhalb von COQ.

12.4.3.2 Reflex-*Bibliothek*

Die Bibliothek REFLEX baut auf YNOT auf und ermöglicht teilweise eine Verifikation [Ric+14]. Allerdings wird REFLEX nicht mehr weiter entwickelt und auch von aktuellen COQ-Versionen nicht mehr unterstützt.

12.4.3.3 Verdi-*Bibliothek*

Wir benutzen die Bibliothek VERDI mit der die Kommunikation verteilter Algorithmen in COQ implementiert und verifiziert werden kann [Wil+15b]. VERDI verfügt über eine gute Dokumentation, die an einer Reihe von Beispielen die Implementierung und Verifikation mit VERDI in COQ erläutert. In [Woo+16] ist außerdem eine größere Fallstudie in VERDI beschrieben: die Implementierung und Verifikation einer Konsensfindung (Raft Protokoll).

Darüber hinaus ist eine Extraktion nach OCAML möglich, sodass die Verifikation erhalten bleibt. Damit der extrahierte verteilte Algorithmus auf einem realen Netzwerk laufen kann, wird er mit einem „Shim“ kombiniert – im Software Engineering ist damit ein kleines Programm gemeint, dass eine Kompatibilität herstellt.

In diesem Fall ist der Shim ein kleines Programm, dass die Aufrufe an der Schnittstelle zwischen einer Komponente und einem Kanal entgegen nimmt und die Parameter entsprechend aufbereitet weitergibt. Er setzt Netzwerkprimitive, wie das Senden und Empfangen von Nachrichten um und ist in OCAML geschrieben. Das heißt der Shim gehört zur Vertrauensbasis bei der Nutzung von VERDI.

Das Team um VERDI arbeitet auch an einer Extraktion zur Programmiersprache `C` [Mul+18], sowie an einer neuen Bibliothek namens DISEL, die eine kompositionale Verifikation ermöglicht [SWT17].

12.4.4 Nutzung der Bibliothek `Verdi`

Wir benutzen die Bibliothek VERDI um die Kommunikation in unserem Netzwerkmodell in COQ zu modellieren und stellen deswegen im folgenden die Bibliothek für diese Nutzung vor.

12.4.4.1 *Implementierung der Kommunikation*

Um die Kommunikation eines verteilten Algorithmus zu implementieren, müssen wir in VERDI zunächst unser verteiltes System definieren. VERDI stellt dafür Netzwerke bereit, allerdings ist damit kein Netzwerk in unserem Sinne gemeint. In VERDI besteht ein Netzwerk aus einer Menge von Komponenten. Der Unterschied ist jedoch, dass es keine Topologie gibt und eine Komponente direkt eine Nachricht zu jeder anderen Komponente schicken kann.

Wie bei uns sind die Netzwerke ID-basiert. Jede Komponente hat also eine eindeutige ID. Darüber hinaus ist jeder Komponente ihr lokaler Zustand zugeordnet und eine Zustandsübergangsfunktion, die sie aufruft, wenn sie eine Nachricht empfängt. Die Funktion berechnet dann einen neuen Zustand und eine Liste zu sendender Nachrichten.

12.4.4.2 *Nachrichtentypen*

VERDI unterscheidet zwischen externen und internen Nachrichten. Komponenten kommunizieren miteinander durch externe Nachrichten. Programme auf einer Komponente kommunizieren miteinander durch interne Nachrichten in dem Modell von VERDI.

12.4.4.3 *Netzwerk in* Verdi

Für ein Netzwerk in VERDI müssen wir die folgenden Informationen bereitstellen:

`Name`: Typ der Komponenten,

`State`: Typ des Zustands einer Komponente,

`Input, Output`: Typ der internen Nachrichten,

`Msg`: Typ der externen Nachrichten,

`initState`: Funktion zur Initialisierung einer Komponente,

`InputHandler`: Funktion, die eine Komponente ausführt, wenn sie eine interne Nachricht erhält,

`NetHandler`: Funktion, die eine Komponente ausführt, wenn sie eine externe Nachricht erhält, sowie

`Nodes`: Menge der Komponenten des Netzwerks.

12.4.4.4 *Zustand eines Netzwerks*

Der Zustand eines Netzwerks umfasst alle Nachrichten, die im Netzwerk unterwegs sind, sowie die lokalen Zustände der Komponenten. In Code-Block 12.12 ist die Implementierung des Zustands `network` mit VERDI dargestellt. Die Nachrichten, die im Netzwerk unterwegs

sind, sind in der Liste `nwPackets` und die lokalen Zustände der Komponenten sind durch die Funktion `nwState` gegeben.

```
Record network :=
    mkNetwork { nwPackets : list packet ;
                nwState : name → state }.
```

Code-Block 12.12: Netzwerkzustand in Verdi.

Der initiale Netzwerkzustand ist definiert durch eine leere Liste von Nachrichten und der Zustand einer jeden Komponente nach Aufruf der Funktion `initState`.

12.4.4.5 *Zustandsübergänge*

Jede (interne oder externe) Nachricht kann jederzeit empfangen werden und wird dann aus dem Netzwerkzustand entfernt. Der Empfänger einer Nachricht führt für interne Nachrichten seinen `InputHandler` und für externe Nachrichten seinen `NetHandler` aus und berechnet so seinen neuen lokalen Zustand sowie seine zu sendenden Nachrichten. Der Netzwerkzustand wird entsprechend aktualisiert.

12.4.4.6 *Programmverifikation in* Verdi

Die Verifikation eines implementierten verteilten Algorithmus funktioniert in Verdi über induktive Invarianten aller erreichbaren Zustände. Für die Verifikation muss also ein Induktionsbeweis über die erreichbaren Zustände des Netzwerks geführt werden, wobei der Induktionsanfang zumeist der initiale Zustand ist. Wie aufwändig dieser Beweis ist, hängt von dem `InputHandler` und `NetHandler` einer jeden Komponente ab, da sich durch sie die Induktionsschritte ergeben.

Als Beispiel für eine Verifikation in Verdi empfehlen wir das `Counter`-Beispiel aus den Beispielen, die die Bibliothek Verdi bereitstellt.

13 FRAMEWORK IN COQ

In diesem Kapitel stellen wir ein für den Beweisassistenten COQ entwickeltes Framework zur formalen Instanzverifikation mit zertifizierenden verteilten Algorithmen vor. Wir haben das Konzept des Frameworks in [VA18] veröffentlicht.

In Abschnitt 13.1 führen wir unser Netzwerkmodell in COQ ein. Wir nutzen hierfür die Bibliotheken GraphBasics und Verdi. In Abschnitt 13.2 geben wir eine Übersicht über das Framework, wobei wir auch die Fallstudien einordnen. In Abschnitt 13.3 vergleichen wir abschließend unser Framework mit einem Framework zur formalen Instanzverifikation für zertifizierende sequentielle Algorithmen, das Rizkallah in ihrer Doktorarbeit entwickelt hat [Riz15].

13.1 NETZWERKMODELL IN COQ

Wir bauen unser Netzwerkmodell in COQ mithilfe der Bibliotheken GraphBasics und Verdi auf. Während GraphBasics uns die Modellierung der Topologie eines Netzwerks ermöglicht, ermöglicht uns Verdi die Modellierung der Kommunikation in einem Netzwerk. Wir vereinen beide Aspekte in einem Modell und berücksichtigen dabei auch die verteilte Architektur eines Checkers (siehe Abschnitt 10.1).

Wir stellen in Abschnitt 13.1.1 die Modellierung der Topologie und in Abschnitt 13.1.2 die der Kommunikation vor. Daraufhin fügen wir die beiden Modellierungen in Abschnitt 13.1.3 zu einem Modell zusammen. In Abschnitt 13.1.4 beschreiben wir abschließend die Initialisierung eines Netzwerks.

13.1.1 Modellierung: Topologie

Wir modellieren Netzwerke als zusammenhängende, ungerichtete Graphen und nutzen dafür entsprechend den Typ `Connected` aus GraphBasics. In Code-Block 13.1 ist die Modellierung der Topologie unseres Netzwerkmodells in COQ dargestellt.

Die physische Komponente (`Component`) entspricht einem Knoten (`Vertex`). Der Graph `g` ist beliebig, aber fest genau wie die Kantenmenge `a` und die Knotenmenge `v`. `Component_list` ist eine Liste, die

die Komponenten des Netzwerks enthält. Das Axiom `Component_prop` modelliert, dass es für jedes Objekt des Typs `Component` auch einen Knoten im Graph gibt.

```
Definition Component := Vertex .
Definition C_set := U_set Component .
Variable a : A_set .
Variable v : C_set .
Variable g : Connected v a .
Axiom Component_pro p :
    forall (c : Component), In c (Component_list v a g) .
```

Code-Block 13.1: Modellierung der Topologie im Netzwerkmodell.

13.1.1.1 *Nachbarn*

GraphBasics stellt zwei Definitionen für die Nachbarschaft bereit: alle eingehenden und alle ausgehenden Nachbarn. Wir benötigen für unser Netzwerkmodell mit bidirektionalen Kanälen einen anderen Begriff der Nachbarschaft. Wir definieren aufbauend auf den beiden Definitionen die Nachbarn einer Komponente in einem ungerichteten, zusammenhängenden Graphen (`Connected`).

Der Code-Block 13.2 zeigt die Definition für Nachbarn.

```
Definition neighbors (g : Connected v a) (c : Component) : C_list :=
    (A_in_neighborhood c (Canal_list v a g)).

Lemma neighbors_connected_prop:
    forall k (g : Connected v a) (c : Component),
        In k (A_out_neighborhood c (Canal_list v a g)) ↔
        In k (A_in_neighborhood c (Canal_list v a g)).
```

Code-Block 13.2: Definition von Nachbarn.

`Canal_list` ist eine Liste, die die Nachrichtenkanäle des Netzwerks enthält – also jeweils zwei gerichtete Kanten für einen bidirektionalen Kanal. `A_in_neighborhood` und `A_out_neighborhood` sind jeweils die eingehenden und ausgehenden Nachbarn. Wir berücksichtigen in der Definition `neighbors` jedoch nur die eingehenden Nachbarn und formulieren deswegen mit dem Lemma `neighbors_connected_prop`, dass dies für einen ungerichteten Graphen keinen Unterschied macht.

13.1.2 Modellierung: Kommunikation

Unser Netzwerkmodell muss die verteilte Architektur eines Checkers umsetzen (siehe Abbildung 10.1 auf Seite 137). Wir modellieren, dass

eine Komponente und ihr Teilchecker jeweils logische Komponenten sind, die gemeinsam auf einer physischen Komponente des Netzwerks liegen. Darüber hinaus müssen auch Teilchecker benachbarter Komponenten miteinander kommunizieren können.

Wir nutzen interne Nachrichten (die VERDI für die Kommunikation logischer Komponenten auf derselben physischen Komponente bietet) für die Kommunikation zwischen einer Komponente und ihrem Teilchecker. Während wir externe Nachrichten (die VERDI für die Kommunikation physischer Komponenten bietet) für die Kommunikation zwischen Teilcheckern benachbarter Komponenten nutzen. Für die Implementierung eines Teilcheckers müssen wir die Funktionen `InputHandler` und `NetHandler` angeben, um vorzugeben, wie interne und externe Nachrichten verarbeitet werden.

VERDI stellt eine Netzwerksemantik für fehlerfreie, asynchrone Kommunikation bereit. Wir nutzen diese Semantik für unser Netzwerkmodell.

13.1.3 Zusammenführung: Topologie und Kommunikation

Wir führen die Modellierung der Topologie mit GRAPHBASICS und die Modellierung der Kommunikation mit VERDI zusammen. Dafür verknüpfen wir den Typ `Component` des Graphen in GRAPHBASICS (siehe Code-Block 13.1) mit dem Typ `Name` einer physischen Komponente in VERDI. Wir integrieren dabei außerdem den Teilchecker einer Komponente als logische Komponente auf der physischen Komponente.

In Code-Block 13.3 ist die Zusammenführung der Modellierung der Topologie und der Kommunikation dargestellt.

```
Inductive Name := Subchecker : Component → Name.
Definition Nodes : list Name :=
    (map Subchecker (Component_list v a g)) .
```

Code-Block 13.3: Zusammenführung der Modellierung der Topologie und der Kommunikation.

Der Konstruktor `Subchecker` einer Komponente des Graphen ordnet der Komponente und ihrem Teilchecker eindeutig eine Netzwerkkomponente mit der ID `Name` in VERDI zu. Mit `Nodes` wird in VERDI die als Menge aller Netzwerkkomponenten angegeben. Die gewählte Definition von `Name` und `Nodes` schränkt die Kommunikation der Netzwerkkomponenten in VERDI auf eine Kommunikation zwischen Nachbarn des Netzwerkgraphen `g` ein.

In Code-Block 13.4 ist außerdem dargestellt, wie wir einen Teilchecker instantiieren.

```
Variable c : Component.
Variable CheckerInstance : Subchecker c.
```

Code-Block 13.4: Instantiierung eines Teilcheckers.

13.1.4 Initialisierung

In Abschnitt 10.1.3 haben wir die Initialisierung für Checker beschrieben, die wir nun auch im Netzwerkmodell in COQ umsetzen.

13.1.4.1 *Wissen über die Nachbarschaft*

Durch eine Initialisierung, die VERDI bereitstellt, wird ein Teilchecker zunächst so initialisiert, dass er seine Nachbarn kennt. Dafür stellt VERDI als Schablone die Funktion `initData` bereit. In unserem Netzwerkmodell berücksichtigt `initData` die Topologie des Netzwerks. Diese Initialisierung ist unabhängig von einem konkreten zertifizierenden verteilten Algorithmus.

13.1.4.2 *Teileingabe, Teilausgabe, Teilzeuge*

Darüber hinaus benötigt ein Teilchecker eine vertrauenswürdige Kopie der Teileingabe der Komponente des Teilcheckers, sowie ein Wissen über den strukturellen Aufbau des verteilbaren Zeugenprädikats. In Code-Block 13.5 ist dieser Teil der Initialisierung mit dem Typ `SubcheckerKnowledge` zu sehen.

```
Record SubcheckerKnowledge : Set := mk_SubcheckerKnowledge {
Sub_inp : list inp;
Dist_predicates : list Predicate
}.

Record SubcheckerInput := mk_SubcheckerInput {
output : list outp ;
witness : list Fact
}.

Inductive Input : Type :=
| ck : SubcheckerKnowledge → Input
| ci : SubcheckerInput → Input.
```

Code-Block 13.5: Initialisierung für Checker im Netzwerkmodell in COQ.

Nachdem eine Komponente ihre Teilausgabe und ihren Teilzeugen berechnet hat, schickt sie beides an ihren Teil-Checker. Da wir den zer-

tifizierenden verteilten Algorithmus als Black-Box sehen und in COQ nicht implementieren, muss es dennoch eine Repräsentation einer Teilausgabe und eines Teilzeugen in COQ geben. Diese Repräsentation gelingt mit dem Typ `SubcheckerInput`. `SubcheckerKnowledge` und `SubcheckerInput` bilden dann gemeinsam den zusätzlichen `Input` zur Initialisierung in VERDI.

13.2 ÜBERSICHT ÜBER DAS FRAMEWORK

Wir geben in diesem Abschnitt eine Übersicht über das Framework für die formale Instanzverifikation zertifizierender verteilter Algorithmen in COQ.

Dafür beschreiben wir in Abschnitt 13.2.1 zunächst das Framework und geben einen Überblick darüber welche Teile implementiert sind. Weiterhin erläutern wir, was für jeden zertifizierenden verteilten Algorithmus getan werden muss, um das Framework zu nutzen,

In Abschnitt 13.2.2 erläutern wir die Vertrauensbasis des Frameworks, also die Frage danach, worauf ein Nutzer trotz formaler Instanzverifikation weiterhin vertrauen muss.

In Abschnitt 13.2.3 diskutieren wir abschließend einige Erweiterungen des Frameworks.

13.2.1 Beschreibung des Frameworks

Das Framework ist in der Abbildung 13.1 dargestellt. Die Nummerierung der Beweisverpflichtungen (BV) entspricht dabei der Nummerierung aus Abschnitt 12.1.2 auf Seite 168.

Wir sehen als Basis das Netzwerkmodell mit der Modellierung der Topologie mit GRAPHBASICS und der Modellierung der Kommunikation mit VERDI, verknüpft wie im vorigen Abschnitt beschrieben. Weiterhin sehen wir auch ein Modell des Algorithmus, über welches wir bisher noch nicht explizit gesprochen haben.

13.2.1.1 *Interface*

Der zertifizierende verteilte Algorithmus ist nicht Teil des Frameworks. Er ist eine Black-Box bei der formalen Instanzverifikation (siehe Kapitel 12). Wir haben im vorigen Abschnitt 13.1.4 die Initialisierung eines Netzwerks in COQ diskutiert. Alle Aspekte, die dabei spezifisch für den konkreten Algorithmus sind, müssen dafür schablonenartig an den entsprechenden Stellen in das Netzwerkmodell integriert werden.

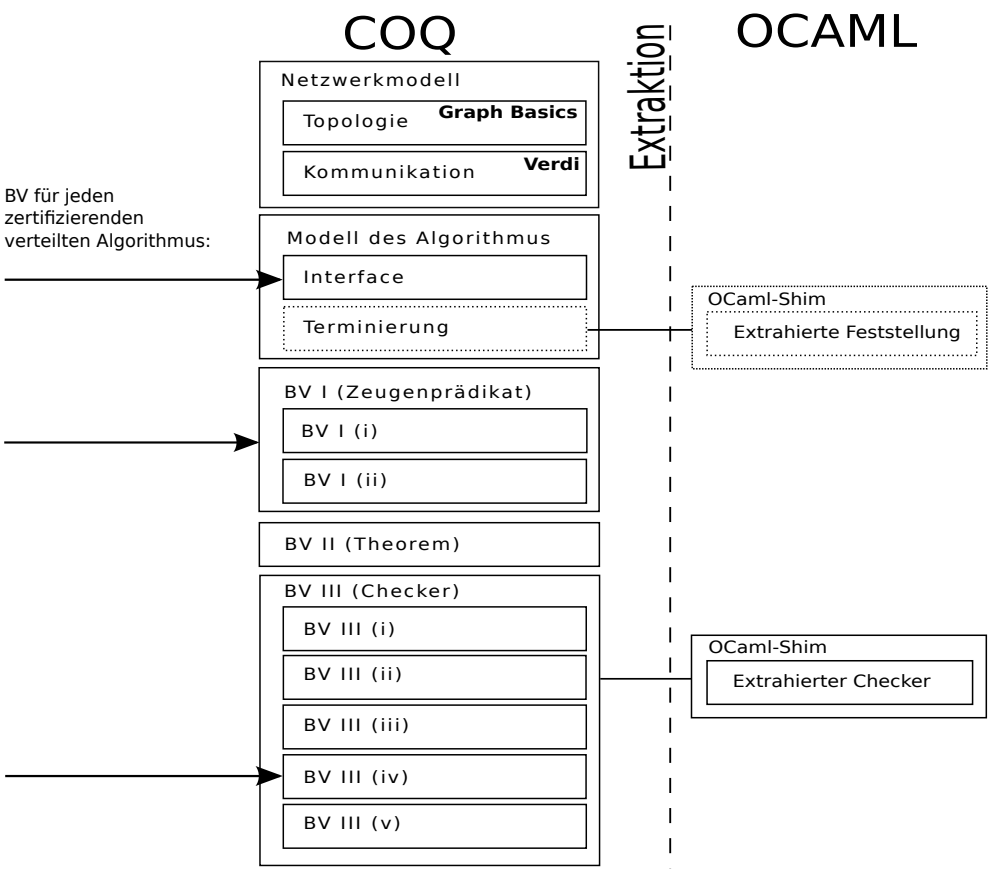


Abbildung 13.1: Framework zur formalen Instanzverifikation in COQ.

Wir bezeichnen diesen Schritt als das Interface bereitstellen und er ist Teil der Modellierung des Algorithmus. Ein weiterer Teil der Modellierung des Algorithmus ist der Aspekt der Terminierung. Wir haben die entsprechenden Boxen mit gepunkteten Rändern dargestellt, da es sich dabei um eine mögliche Erweiterung des Frameworks handelt. Wir kommen darauf in Abschnitt 13.2.3 zu sprechen.

13.2.1.2 *Spezifische Beweisverpflichtungen*

In der Abbildung 13.1 sehen wir auch, dass einige Beweisverpflichtungen einen eingehenden Pfeil haben. Diese Beweisverpflichtungen sind *spezifisch* für jeden zertifizierenden verteilten Algorithmus. Das Zeugenprädikat mit seiner Zeugeneigenschaft und seiner Verteilungseigenschaft unterscheidet je nach Algorithmus beziehungsweise zu lösendem Problem. Im Falle des Checkers unterscheiden sich die Entscheidungsprozeduren der lokalen Prädikate.

Wir haben anhand dieses Frameworks drei Fallstudien durchgeführt, die sich mit den spezifischen Beweisverpflichtungen beschäftigen: die Fallstudie Kürzesten-Pfade (siehe Abschnitt 12.3) und die Fallstudien der Leader Election und des Bipartitheitstest, die wir in Kapitel 15 vorstellen.

13.2.1.3 *Allgemeine Beweisverpflichtungen*

Neben den spezifischen Beweisverpflichtungen gibt es auch *allgemeine* – also solche, die einmal bewiesen, für alle zertifizierenden verteilten Algorithmen gelöst sind.

Hierunter fallen alle Aufgaben eines Checkers mit Ausnahme der Entscheidung der lokalen Prädikat. In der Abbildung ist das Framework für eine lokale Konsistenzprüfung (siehe Abschnitt 10.4) dargestellt. Dadurch ist zum Beispiel auch das Theorem zur lokalen Konsistenz eine Beweisverpflichtung (BV II).

Wir haben für das Framework zwei implementierte und verifizierte Konsistenzprüfungen bereitgestellt: die lokale Konsistenzprüfung zusammenhängender Zeugen und die Konsistenzprüfung für beliebige Zeugen. Wir stellen beide in Kapitel 14 vor.

13.2.1.4 *Modellierung, Theorembeweisen, Programmverifikation*

Unser Framework ist so aufgebaut, dass es vollständig in COQ implementiert werden kann. Dabei bedienen wir uns sowohl der Möglichkeiten der Modellierung, des Theorembeweisens und der Programmverifikation in COQ. Zum Beispiel modellieren wir zusammenhängende Zeugen, beweisen das Theorem zur lokalen Konsistenzprüfung und verifizieren eine implementierte lokale Konsistenzprüfung.

Bei der Programmverifikation ist auch eine Extraktion implementierter und verifizierter verteilter Algorithmen nach OCaml möglich (siehe Kapitel 12).

13.2.1.5 *Framework als Proof-of-concept*

Das hier aufgezeigte Framework ist als Proof-of-concept zu verstehen. Wir haben eine Methode der formalen Instanzverifikation für zertifizierende verteilte Algorithmen vorgestellt. Weiterhin haben wir aufgezeigt, wie diese Methode in COQ unter Nutzung zweier Bibliotheken gelingt.

Wir zeigen außerdem anhand von drei Fallstudien zertifizierender verteilter Algorithmen exemplarisch die Lösung spezifischer Beweisverpflichtungen im Framework. Die Zukunftsvision ist ein Aufbau einer Bibliothek zertifizierender verteilter Algorithmen, für die im Framework eine formale Instanzverifikation vorliegt. Der Aufbau einer

solchen Bibliothek ist ein aufwändiges Projekt für ein größeres Team. Die Fallstudien machen hier jedoch einen Anfang und zeigen auf, wie eine solche Bibliothek aufgebaut werden kann.

Darüber hinaus zeigen wir anhand zweier Konsistenzprüfungen exemplarisch die Lösung allgemeiner Beweisverpflichtungen im Framework, für die eine Programmverifikation nötig ist. Wir haben nicht alle allgemeinen Beweisverpflichtungen des Frameworks gelöst. Unsere Auswahl ist jedoch so gewählt, dass sie Lösungsansätze für die offenen Beweisverpflichtungen bietet. Zum Beispiel haben wir die Evaluation weder implementiert noch verifiziert. Eine Implementierung und Verifikation würde jedoch sehr ähnlich aussehen, wie bei der allgemeinen Konsistenzprüfung.

13.2.2 Vertrauensbasis

Wir fassen die Vertrauensbasis des Frameworks zusammen [Tho84]. Ein Nutzer eines zertifizierenden verteilten Algorithmus, für den eine formale Instanzverifikation mit dem beschriebenen Framework vorliegt, muss auf die folgenden Punkte vertrauen:

- die Korrektheit des Beweisassistenten COQ,
- Korrektheit des Compilers für OCaml,
- Korrektheit des OCaml-Shim für die Nutzung mit Verdi, sowie
- die korrekte Funktionsweise der Hardware bei der Beweisführung, der Kompilierung und der Ausführung der Checker.

An dieser Stelle ist insbesondere Interessant, dass COQ ist nach dem Kriterium von de Brujn aufgebaut ist [Geu09]: Das heißt bei der Nutzung von COQ müssen wir nur auf die Korrektheit eines „kleinen“ Kerns der Implementierung vertrauen. Klein bedeutet für den Beweisassistenten COQ um die fünfzehn Regeln zur Typprüfung, mit denen die Beweisprüfung erfolgt.

13.2.3 Erweiterungen des Frameworks

Wir diskutieren einige Erweiterungen des Frameworks, wie eine implementierte und verifizierte Feststellung der Terminierung, eine verifizierte Fehlertoleranz für Checker und eine verifizierte Synthese für Teilchecker.

13.2.3.1 *Terminierung*

Wie für zertifizierende sequentielle Algorithmen, geben wir mit einem zertifizierenden verteilten Algorithmus die Garantie der Korrektheit,

für den Fall, dass der Checker akzeptiert. Wenn der zertifizierende verteilte Algorithmus nicht terminiert, so beginnt der Checker auch nie mit seiner Prüfung. Bei einer fehlerhaften Feststellung der Terminierung bewertet der Checker die Ausgabe mit dem Zeugen, die er erhalten hat.

In Kapitel 5 haben wir bereits eine generische Terminierungsfeststellung aus [Ray13] skizziert. Eine Erweiterung des Frameworks könnte es deswegen sein, diese in COQ zu implementieren und verifizieren.

13.2.3.2 *Fehlertolerante Checker*

VERDI stellt eine Netzwerksemantik bereit, mit dem ein verteilter Algorithmus bei der Extraktion fehlertolerant gestaltet werden kann (`VerdiSystemsTransformers`). Das resultierende Programm ist dann zum Beispiel tolerant gegen Fehler, wie Nachrichtenverlust oder ausfallende Komponenten.

Es gelten dabei die gleichen Garantien, wie bei der Extraktion, wie mit der fehlerfreien Semantik: verifizierte Eigenschaften bleiben erhalten. Das heißt die in VERDI verifizierte Fehlertoleranz führt zu einem extrahierten fehlertoleranten Checker, der auf einem realen Netzwerk läuft.

13.2.3.3 *Verifizierte Synthese von Teilcheckern*

Die Beweisverpflichtung zur Entscheidung der lokalen Prädikate ist spezifisch für jeden zertifizierenden verteilten Algorithmus. Im Bereich des Monitoring wiederum gibt es viele Arbeiten zur automatische Synthese eines Monitors [MP16; Fra+17; Shi+17; KS18]. Dabei ist die Eingabe für das Programm zur Synthese die temporal-logische Formel, deren Einhaltung der Monitor überwacht. Einige Synthese-Programme sind darüber hinaus auch formal verifiziert.

Wir haben eine allgemeine Übertragung des Konzepts zertifizierender sequentieller Algorithmen angestrebt und deswegen die Logik für lokale Prädikate nicht eingeschränkt. In unseren Fallstudien (siehe Kapitel 9) sind jedoch für Entscheidungsprozeduren der lokalen Prädikate lediglich einfache Arithmetik (Addition, Subtraktion ganzzahliger Zahlen), sowie Vergleiche nötig.

Für eine eingeschränkte Logik für die lokalen Prädikate könnte deswegen eine Erweiterung des Frameworks eine verifizierte Synthese der Entscheidungsprozeduren sein. Das Synthese-Programm kann dabei in COQ implementiert und verifiziert werden, sodass sich an der Vertrauensbasis nichts ändert.

13.2.3.4 *Automatisierungen beim Theorembeweisen*

Weitere spezifische Beweisverpflichtungen ergeben sich zum Zeugenprädikat. In den Fallstudien dieser Arbeit kommen beim Theorembeweisen nur die Automatisierungen durch die Taktiken, die COQ vorgibt zum Einsatz.

Eine andere Option ist es, benutzerdefinierte Taktiken für COQ zuschreiben, sodass einige Teile der Beweisführung automatisiert werden. In [Vö13] wird zum Beispiel eine Bibliothek mit Taktiken zur teilweisen Automatisierung für Beweise über ein Speichermodell der Sprache `C` aufgebaut. COQ stellt mächtige Werkzeuge für zur Entwicklung benutzerdefinierter Taktiken bereit [Chl13].

13.3 VERGLEICH MIT RIZKALLAHS FRAMEWORK FÜR ZERTIFIZIERENDE SEQUENTIELLE ALGORITHMEN

Rizkallah hat im Rahmen ihrer Doktorarbeit [Riz15] ein Framework für formale Instanzverifikation mit zertifizierenden *sequentiellen* Algorithmen entwickelt [Alk+11; Alk+14; NRM14; Riz14]. Wir vergleichen unser Framework mit diesem Framework.

In Abschnitt 13.3.1 vergleichen wir das methodische Vorgehen und in Abschnitt 13.3.2 die Umsetzung im jeweiligen Framework.

13.3.1 Methodik

Da ein zertifizierender verteilter Algorithmus ein Spezialfall eines zertifizierenden sequentiellen Algorithmus ist (siehe Kapitel 8), haben wir bei der formalen Instanzverifikation mit einem zertifizierenden verteilten Algorithmus alle Beweisverpflichtungen wie bei einem zertifizierenden sequentiellen Algorithmus. Darüber hinaus fallen jedoch weitere Beweisverpflichtungen an.

So muss für ein verteilbares Zeugenprädikat zusätzlich die Verteilungseigenschaft gezeigt werden. Weiterhin gliedert sich die Entscheidung des Zeugenprädikats in die Entscheidung der lokalen Prädikate und die globale Evaluation im Netzwerk. Durch den verteilten Zeugen muss ein Checker außerdem die Aufgabe der Konsistenzprüfung übernehmen.

13.3.2 Umsetzung in einem Beweisassistenten

Rizkallah hat ihr Framework in dem Beweisassistenten Isabelle umgesetzt, während wir unser Framework in COQ umsetzen. Wir vergleichen die Umsetzung beider Frameworks anhand der Aspekte Modellierung, Theorembeweisen und Programmverifikation.

13.3.2.1 *Modellierung*

Im Vergleich zu Rizkallahs Framework gibt es in unserem Framework im Allgemeineren einen größeren Aufwand bei der Modellierung. Das liegt daran, dass wir uns mit verteilten Algorithmen beschäftigen und deswegen zusätzlich noch ein Netzwerkmodell in einem Beweisassistenten benötigen. Ein Vergleich zwischen dem Framework dieser Arbeit und Rizkallahs Framework bietet sich für den Aspekt der Modellierung also nicht an.

13.3.2.2 *Theorembeweisen*

Für einige Beweisverpflichtungen ist es nötig ein Theorem zu beweisen. Für die Umsetzung dieser Beweisverpflichtungen sind, unserer Ansicht nach, beide Beweisassistenten im Allgemeinen gleichermaßen geeignet. Eine Entscheidung für den einen oder anderen Beweisassistenten ist an dieser Stelle zum einen eine Frage des Geschmacks und zum anderen eine Frage des Vorwissens der Personen, die die Beweise führen.

Darüber hinaus ist es interessant zu betrachten, welche Bibliotheken in den Beweisassistenten jeweils bereits vorhanden sind. Sowohl Isabelle, als auch COQ haben eine breite Nutzerbasis und es gibt für beide diverse Bibliotheken. Je nach Bereich kann sich jedoch mal der eine oder der andere Beweisassistent anbieten.

Bei verteilten Algorithmen ist zu erwarten, dass für die Zertifizierung auch Theoreme über Graphen gezeigt werden müssen, da das Netzwerk bereits als Graph modelliert ist. Wir haben dafür Bibliothek GraphBasics genutzt, die nicht sonderlich umfangreich ist. Wir mussten sie im Rahmen unserer Nutzung bereits erweitern. Dennoch ist aus unserer Sicht GraphBasics für unsere Zwecke die geeignetste Bibliothek für Graphen in COQ.

Obwohl für sequentielle Algorithmen nicht zu erwarten ist, dass Graphen bei der Zertifizierung eine übergeordnete Rolle spielen, können wir dennoch einen Vergleich ziehen. Das liegt daran, dass Rizkallah als Fallstudien zertifizierende Graphalgorithmen gewählt hat. Dafür hat Rizkallah eine Bibliothek genutzt, die von Noschinski angelehnt an GraphBasics im Rahmen der Arbeiten zum Framework Frameworks

für Isabelle entwickelt wurde [Nos14]. Die Situation in Isabelle für Graphen ist folglich mit der in COQ vergleichbar.

13.3.2.3 *Programmverifikation*

Für die Programmverifikation hat Rizkallah Isabelle mit weiteren Werkzeugen kombiniert. Der Grund dafür ist, dass es in Isabelle keine Programmextraktion, wie in COQ gibt.

Rizkallah zeigt auf, wie Isabelle und VCC kombiniert werden können – VCC ist eine Umgebung zur Verifikation von Programmen in der Sprache `C` [Mos11]. Durch die Kombinationen erhält sie eine weitere Beweisverpflichtung zur Konsistenz der Formalisierung in beiden Werkzeugen. Weiterhin vergrößert sich die Vertrauensbasis durch die Nutzung eines weiteren Werkzeugs.

Um diese beiden Probleme anzugehen, hat Rizkallah Isabelle mit AutoCorres/simpl kombiniert – AutoCorres/simpl ist eine Formalisierung eines Speichermodells für einen eingeschränkten Teil der Sprache `C` in Isabelle. Die zusätzliche Beweisverpflichtung zur Konsistenz entfällt dadurch und die Vertrauensbasis wird nicht vergrößert.

Allerdings müssen in Isabelle für die Programmverifikation Beweise geführt werden, in denen über die Ausführung eines `C`-Programms mit dem formalisierten Speichermodell argumentiert wird. Das ist unserer Ansicht nach weniger elegant als über Programme in COQ zu argumentieren, die in in der eigenen funktionalen Programmiersprache geschrieben sind. In COQ sind damit die Programmierung und Beweisführung direkt miteinander verbunden.

14 KONSISTENZPRÜFUNG IN COQ

Wir haben in Abschnitt 10.4 zwei Konsistenzprüfung eines Zeugen vorgestellt: eine lokale Konsistenzprüfung zusammenhängender Zeugen und eine allgemeine Konsistenzprüfung beliebiger Zeugen. In diesem Kapitel diskutieren wir die Implementierung und Verifikation beider Konsistenzprüfungen in COQ im Rahmen des Frameworks für die formale Instanzverifikation.

In Abschnitt 14.1 stellen wir die lokale Konsistenzprüfung und in Abschnitt 14.2 die allgemeine Konsistenzprüfung vor. Die lokale Konsistenzprüfung ist in [Aki18] veröffentlicht und die allgemeine Konsistenzprüfung in [Bol19]. Bei den beiden Arbeiten handelt es sich um studentische Abschlussarbeiten, die von der Autorin der vorliegenden Arbeit betreut wurden.

14.1 LOKALE KONSISTENZPRÜFUNG ZUSAMMENHÄNGENDER ZEUGEN

Wir stellen Ausschnitte aus der Implementierung und Verifikation der lokalen Konsistenzprüfung in COQ vor.

In Abschnitt 14.1.1 besprechen wir die Definition des Begriffs der Konsistenz in COQ. In Abschnitt 14.1.2 beschreiben wir die Implementierung der lokalen Konsistenzprüfung und in Abschnitt 14.1.3 deren Verifikation. Abschließend diskutieren wir in Abschnitt 14.1.4 die Formalisierung des grundlegenden Theorems 7.3.4 der lokalen Konsistenzprüfung in COQ.

14.1.1 Definition der Konsistenz in COQ

Wir formalisieren den Begriff der Konsistenz in COQ kleinschrittig in aufeinander aufbauenden Definitionen. Dadurch können einzelne Aspekte besser wiederverwendet werden.

14.1.1.1 *Überlappende Teilzeugen*

In Abschnitt 7.2.1 haben wir überlappende Zeugen definiert. In COQ benutzen wir den Begriff der Überlappung auch kanonisch herunter

gebrochen für Teilzeugen. In Code-Block 14.1 ist die Formalisierung des Begriffs der Überlappung in COQ dargestellt.

```
Definition overlap (w1 w2 : Witness) (k : Key) :=
(option nat * option nat)
    match (get_pos w1) , (get_pos w2) with nat) :=
        | None, None => ( None, None )
        | x, None => ( None, None )
        | None, y => ( None, None )
        | x, y => (x, y )
    end.

Definition overlap_in_var (w1 w2 : Witness) (k : Key):=
    exists (x y : nat), (Some x, Some y) = overlap w1 w2.
```

Code-Block 14.1: Überlappende Zeugen in COQ.

Die Funktion `overlap` hat als Argumente zwei Teilzeugen `w_1` und `w_2`, sowie eine Variable `k`. Wenn die Variable in beiden Teilzeugen vorkommt, ist der Rückgabewert ein Tupel `(x,y)`, das die jeweilige Position der Variable in dem jeweiligen Teilzeugen angibt. Zusätzlich formalisieren wir den Begriff der Überlappung zwischen zwei Teilzeugen in einer Variable als Prädikat.

14.1.1.2 *Konsistenz*

In Code-Block 14.2 ist die Definition der Konsistenz zweier Teilzeugen in COQ dargestellt.

```
Definition consistent (w1 w2 : Witness) :=
    forall k, overlap_in_var w1 w2 k →
        eq (findValue w1 (get_pos w1 k)) (findValue w2 (get_pos w2 k)).
```

Code-Block 14.2: Begriff der Konsistenz in COQ.

Die Konsistenz wird auf Basis der Überlappung definiert: zwei Teilzeugen sind konsistent, wenn sie in allen überlappenden Variablen übereinstimmen. Um die Beweise übersichtlicher zu gestalten, formalisieren wir zusätzlich die Konsistenz von Teilzeugen für einzelne Variablen (nicht dargestellt).

Darüber hinaus beweisen wir, dass die Definitionen sich entsprechend zu einander verhalten. Solch ein Theorem, das Definitionen in Beziehung setzt, ist sinnvoll, um die eigene Modellierung zu überprüfen. Wie auch auf Papier stellt sich in COQ immer die Frage, ob wir auch das definiert haben, was wir wollen. Wir verzichten an dieser Stelle auf die Darstellung in COQ.

14.1.1.3 *Konsistenz in Nachbarschaften*

In Code-Block 14.3 ist die Definition der Konsistenz einer Nachbarschaft als Prädikat über einer einzelnen Komponente, sowie die Definition der Konsistenz in allen Nachbarschaften als Prädikat über dem gesamte Netzwerk `con` in COQ dargestellt.

```
Definition neighbourhood_consistent (comp : Component) :=
    forall (comp1 : Component), In comp1 (neighbors g comp) →
consistent_in_all_var (getWitness comp) (getWitness comp1).

Definition all_neighbourhoods_consistent (con : Connected v a) :=
    forall (comp : Component), (v comp) →
        neighbourhoods_consistent comp.
```

Code-Block 14.3: Konsistenz in Nachbarschaften in COQ.

14.1.1.4 *Zusammenhängender Zeuge*

Die lokale Konsistenzprüfung eignet sich zur Prüfung der Konsistenz zusammenhängender Zeugen. Wir definieren in COQ deswegen auch zusammenhängende Zeugen. In Code-Block 14.4 ist die Definition in COQ dargestellt.

```
Definition witness_connected_in_var (con : Connected v a) (k : Key) :=
    forall (comp1 comp2 : Component), (v comp1) → (v comp2) →
        {vl : V_list & {el : E_list & {p : Path v a comp1 comp2 vl el
        & path_connected_in_var comp1 comp2 vl el p k }}}.

Definition witness_connected (con : Connected v a) :=
    forall k, witness_connected_in_var con k.
```

Code-Block 14.4: Definiton zusammenhängender Zeuge in COQ.

Dabei definieren wir zunächst den Zusammenhang in einer Variable für die Teilzeugen zweier Komponenten `comp1` und `comp2`. Darauf aufbauend definieren wir dann einen zusammenhängenden Zeugen im Netzwerk `con`.

14.1.2 Implementierung in Verdi

Für die Konsistenzprüfung entscheidet jeder Teilchecker einer Komponente, ob die Teilzeugen in seiner Nachbarschaft konsistent sind. Dafür müssen benachbarte Teilchecker ihre Teilzeugen austauschen.

In Code-Block 14.5 ist die mit VERDI implementierte lokale Konsistenzprüfung dargestellt.

```
Record Data := mkData {
    myWitness : Witness;
    variables : list A;
    nbrslist : list Component;
    consistent : bool;
    initialized : bool;
    initialized_1 : bool;
    initialized_2 : bool;
}.

Definition InputHandler (me : Name) (c : Input) (state : Data) :=
match me, c with
    | Subchecker x, SubcheckerKnowledge ⇒
       if (initialized == false)
       then variables := SubcheckerKnowledge.(variables)
       initialized_1 := true
       if (initialized_1 && initialized_2) then
           initialized := true
    | Subchecker x, SubcheckerInput ⇒
       if ( initialized == false ) then
           myWitness := SubcheckerInput.(Witness)
           sendToAllNeighbors (myWitness)
           initialized_2 := true
           if (initialized_1 && initialized_2) then
               initialized := true

Definition NetHandler (me: Name)(src: Name)(m: Msg)(state: Data):=
match m with
    | Subchecker message Witness ⇒ if (initialized == true) then
       nbrslist := remove src state.(nbrslist)
       consistent := state.(consistent) &&
       Consistency_Nbr (Witness)
```

Code-Block 14.5: Implementierung der lokalen Konsistenzprüfung in COQ mmit Verdi.

Mit `Data` wird in VERDI der lokale Zustand einer Netzwerkkomponente beschrieben. In diesem Fall handelt es sich um den lokalen Zustand eines Teilcheckers mit den folgenden Einträgen:

`myWitness`: Teilzeuge der Komponente des Teilcheckers,

`variables`: alle Variablen über die der Teilzeuge eine Aussage macht,

`nbrslist`: die Nachbarn des Teilcheckers, von denen noch kein Teilzeuge erhalten wurde,

`consistent`: Status der Konsistenz der Teilzeugen in der Nachbarschaft,

`initialized` : gemeinsam mit den anderen beiden booleschen Werten zur Initialisierung dient `initialized` zur schrittweisen Initialisierung des Teilcheckers (siehe Abschnitt 13.1.4).

Der lokale Zustand enthält noch ein paar weitere Informationen, die die Verifikation vereinfachen, aber für die Implementierung unwichtig sind. Die Implementierung zeigt, wie ein Teilchecker interne und externe Nachrichten verarbeitet und damit in einen neuen lokalen Zustand übergeht.

Die Funktion `InputHandler` verarbeitet die internen Nachrichten. Wir nutzen den `InputHandler` zur Initialisierung, wie in Abschnitt 13.1.4 beschrieben. Die Argumente der Funktion sind die ID des Teilchecker, die er sich mit seiner Komponente teilt, die interne Nachricht, sowie der lokale Zustand des Teilcheckers.

Darüber hinaus lassen wir eine Teilchecker im Zuge der Abarbeitung des `InputHandlers` den Teilzeugen seiner Komponente als externe Nachricht an benachbarte Teilchecker schicken. Dies dient als „Bootstrapping"-Mechanismus, um den zur Konsistenzprüfung benötigten Nachrichtenversand im Netzwerk in Gang zu setzen.

Mit der Funktion `NetHandler` wird die Verarbeitung externer Nachrichten beschrieben. Dabei führt ein Teilchecker buch, von welchem Nachbar er bereits einen Teilzeugen erhalten hat und ob die Teilzeugen konsistent ist. Das Prüfung der Konsistenz zwischen einem empfangen Teilzeugen und dem Teilzeugen der eigenen Komponente geschieht in der Funktion `Consistency_Nbr`, auf deren Darstellung wir hier verzichten.

14.1.3 Programmverifikation

Für die Verifikation der Implementierung beweisen wir in COQ, dass wenn ein Teilchecker die Konsistenz in seiner Nachbarschaft entscheidet, das dann für alle Teilzeugen der Nachbarschaft gilt, dass diese konsistent sind mit dem Teilzeugen der Komponente des Teilcheckers. Für eine bessere Lesbarkeit verzichten wir an dieser Stelle auf eine Darstellung in COQ, die viele technische Details zur Implementierung, enthält.

Wir wählen stattdessen eine Formulierung des Theorems, die einen etwas größeren Abstraktionsgrad darstellt:

Theorem 14.1.1 (Programmverifikation in COQ). *Für alle Teilchecker* `C` *des Netzwerks gilt, dass wenn*

(i) `initialized(C) = true`,

(ii) `nbrslist(C) = empty` *und*

(iii) `consistent(C) = true`

erfüllt sind, dann gilt `neighbourhood_consistent(C)`.

Dabei benutzen wir in den Bedingungen (i) – (iii) Variablen aus dem Zustand eines Teilcheckers (siehe Code-Block 14.5). Wir beweisen das Theorem zur Programmverifikation in COQ mithilfe eigener Lemmata, in denen wir zunächst Invarianten über den lokalen Zustand eines Teilcheckers bei der Ausführung zeigen. Ein Beispiel für eine solche Zustandsinvariante ist, dass insgesamt alle Nachbarn eines Teilcheckers entweder noch in seiner Liste `nbrslist` vorkommen oder aber er bereits einen Teilzeugen des entsprechenden Nachbars erhalten hat.

14.1.4 Theorem zur lokalen Konsistenzprüfung

Die lokale Konsistenzprüfung zusammenhängender Zeugen basiert auf dem Theorem 7.3.4 (siehe Seite 99): ein zusammenhängender Zeuge ist konsistent, wenn die Teilzeugen aller Nachbarschaften konsistent sind.

In Code-Block 14.6 ist das Theorem in COQ dargestellt.

```
Theorem local_consistency_checkable:
    witness_connected g →
    all_neighbourhoods_consistent →
    witness_consistent g.
```

Code-Block 14.6: Theorem 7.3.4 in COQ.

Wir verzichten an dieser Stelle darauf, den Beweis in COQ zu erläutern. Im Vergleich zu dem Beweis, den wir für das Theorem in dieser Arbeit auf Papier geführt haben, ist er detaillierter. Zum Beispiel haben wir in dem Beweis auf Papier argumentiert, dass sich die Konsistenz in einem Netzwerk über den Zusammenhang eines Zeugen transitiv fortsetzt. In COQ haben wir für dieses Argument zwei eigene Lemmata formuliert und bewiesen. Der Beweis in COQ ist entsprechend aufwändiger als auf Papier. Die dabei entwickelten Lemmata sind so allgemeiner Natur, dass sie im Rahmen des Framework wiederverwendet werden können.

14.1.4.1 *Prüfung des Zusammenhangs*

Die lokale Konsistenzprüfung setzt einen zusammenhängenden Zeugen voraus. In Abschnitt 10.4 haben wir einen entsprechenden Algorithmus beschrieben. Wir haben ihn jedoch nicht in COQ implementiert.

14.2 KONSISTENZPRÜFUNG BELIEBIGER ZEUGEN

Wir stellen Ausschnitt der Implementierung und Verifikation der allgemeinen Konsistenzprüfung in COQ vor.

In Abschnitt 14.2.1 beschreiben wir die Implementierung der Konsistenzprüfung und in Abschnitt 14.2.2 deren Verifikation.

14.2.1 Implementierung

Für die allgemeine Konsistenzprüfung setzen wir einen Spannbaum im Netzwerk voraus. Der Spannbaum kann dabei zum Beispiel dem Evaluationsbaum entsprechen. Wir haben die allgemeine Konsistenzprüfung in Abschnitt 10.4 beschrieben.

In Code-Block 14.7 ist der verteilte Algorithmus in VERDI implementiert dargestellt.

```
Definition InputHandler (me: Subchecker) (data: Data) :
(list Output) * Data * list (Subchecker * Witness) :=
match (children me) with
| [] ⇒ ([], data, [( parent me, (witness_list data))])
| _ ⇒ ([], data, [])
end.

Definition NetHandler (me : Subchecker) (src: Subchecker)
(msg : Witness) (data: Data) :
(list Output) * Data * list (Subchecker * Witness) :=
match (child_todo data) with
    | [] ⇒ ([witness_list_consistent
        (witness_list data)], data, [])
    | [c] ⇒
        ([], (mkData ((witness_list data) ++ msg) []),
        [( parent me, (witness_list data ++ msg))])
    | _ ⇒ ([], (mkData ((witness_list data) ++ msg)
        (remove_child src (child_todo data))), [])
end.
```

Code-Block 14.7: Implementierung der Konsistenzprüfung in COQ mit Verdi.

Jeder Teilchecker hat eine Liste `witness_list` von Variablenbelegungen. Anfangs sind dort genau die Belegungen des Teilzeugen der eigenen Komponente enthalten. Von den Blättern des Spannbaums startend, schickt jeder Teilchecker seine Liste an seinen Elternknoten. Wir nutzen die Funktion `InputHandler`, um die Konsistenzprüfung bei den Blättern zu starten.

Erhält ein Teilchecker eine Liste von einem seiner Kinder, so nimmt er die neuen Variablenbelegungen in seine Liste auf. Erst wenn er eine Liste von jedem Kind erhalten hat, schickt er sie an seinen Elternknoten weiter. Die Funktion `child_todo` hilft dabei, über die Nachrichten der Kinder buchzuführen. Die Wurzel prüft die Konsistenz anhand der vollständigen `witness_list` und informiert alle über den Spannbaum. Die Prüfung der Konsistenz ist in der Funktion `witness_list_consistent` umgesetzt.

14.2.2 Programmverifikation

Für die Programmverifikation muss gezeigt werden, dass wenn die Wurzel des Spannbaums entscheidet, dass die Teilzeugen konsistent sind, dass der Zeuge dann konsistent ist. In Code-Block 14.8 ist das Theorem in COQ dargestellt.

```
Theorem output_true_witness_consistent:
forall (net : Network) (tr : Trace) (c : Component),
    In (c, inr [true]) tr →
    witness_consistent.
```

Code-Block 14.8: Theorem der Programmverifikation in COQ

Um dieses Theorem der Programmverifikation in COQ zu beweisen, beweisen wir einige Lemmata. Zum Beispiel, dass jeder Teilchecker nur Nachrichten an die Komponenten schickt, die ein Elternknoten im Spannbaum ist. Darüber hinaus, dass jeder Teilchecker außerdem nur Nachrichten von Komponenten erhält, die seine Kinder im Spannbaum sind oder auch dass nur Kinder einer Komponente in der Liste `child_todo` des Teilcheckers dieser Komponenten sind.

Einige Lemmata sind jedoch noch offen geblieben, wie zum Beispiel: für zwei Nachrichten, die noch nicht abgearbeitet wurden, ist nie der Sender der einen Nachricht auch der Empfänger der anderen Nachricht. Die Eigenschaft gilt, da Nachrichten nur entlang der Elternrelation des Spannbaum gesendet werden und damit sichergestellt ist, dass es solche Kreise nicht gibt. Die Eigenschaft ist wichtig um zu zeigen, dass Teilchecker nicht gegenseitig aufeinander warten und der verteilte Algorithmus somit in einen Deadlock gerät.

Im Vergleich zur lokalen Konsistenzprüfung gestaltet sich die Programmverifikation aufwändiger, was daran liegt, dass hier nicht nur über Nachbarschaften argumentiert werden muss.

15 WEITERE FALLSTUDIEN ZUR FORMALEN INSTANZVERIFIKATION IN COQ

Wir stellen in diesem Kapitel Ausschnitte zweier Fallstudien zur formalen Instanzverifikation vor. Wir verzichten darauf die vollständige Implementierung in COQ zu zeigen. Unser Ziel ist es, den Leser:innen einen Eindruck beim Vorgehen der formalen Instanzverifikation zu vermitteln.

In Abschnitt 15.1 zeigen wir eine formale Instanzverifikation für eine zertifizierende Leader-Election. Unsere Ergebnisse hierzu sind in [VA17] veröffentlicht. Wir haben in Abschnitt 9.7 eine zertifizierende Leader-Election vorgestellt, die dabei Ausgangspunkt für die Umsetzung in COQ ist.

In Abschnitt 15.2 illustrieren wir unsere Methode der formale Instanzverifikation an dem zertifizierenden verteilten Bipartitheitstest (siehe Abschnitt 9.8). Die Fallstudie ist im Rahmen der Arbeit [Bol19] entstanden – eine Diplomarbeit, die die Autorin dieser Arbeit betreute.

15.1 FORMALE INSTANZVERIFIKATION DER LEADER-ELECTION

In Abschnitt 15.1.1 stellen wir die Formalisierung des Zeugenprädikats in COQ vor und in Abschnitt 15.1.2 die Implementierung und Verifikation des Checkers.

15.1.1 Zeugenprädikat

Wir erinnern daran, dass die Zertifizierung der Leader Election einen Spannbaum nutzt, dessen Wurzel der gewählte Leader ist. Der Teilzeuge einer jeden Komponente ist aufgebaut aus:

- der Distanz der Komponente zu ihrem gewählten Leader im Spannbaum,
- ihrem Elternknoten im Spannbaum,

- der Distanz des Elternknotens zu dessen gewähltem Leader und
- die gewählten Leader aller Nachbarn.

In Code-Block 15.1 ist das lokale Prädikat der Komponenten für die Zertifizierung der Leader-Election in COQ dargestellt.

```
        Definition gamma_i (i:Component)(leader(i):Component)
        (distance(i) nat)(parent(i):Component)(leader(parent(i)):
            Component)
(distance_parent_i:nat): Prop :=
    leader(i) <> i ∧
    a (A_ends i parent(i)) ∧
    distance(i) = distance(parent(i)) + 1 ∧
    In parent(i) (neighbors g i) ∧
    (forall (k: Component), In k (neighbors g i)
        → leader(k) = leader(i)).
```

Code-Block 15.1: Lokales Prädikate bei der Zertifizierung der Leader Election in COQ.

Zur Vereinfachung haben wir die Klausel für den Fall der Wurzel weggelassen. Für alle anderen Komponenten gilt, dass sie sich nicht selbst als Leader gewählt haben.

Wir nehmen an, dass das lokale Prädikat für jede Komponente erfüllt ist und formulieren unter dieser Prämisse in COQ die Zeugeneigenschaft. In Code-Block 15.2 ist das Theorem für die Zeugeneigenschaft in COQ, ohne die Annahmen des Moduls dargestellt. Wir haben dieses Vorgehen bereits bei der Verfikation der Zeugeneigenschaft unserer ersten Fallstudie gesehen (siehe Kapitel 12).

```
Theorem witness_property:
exists (l: Component), v l ∧
forall (x: Component)(prop1: v x), leader(x) = l.
```

Code-Block 15.2: Zeugeneigenschaft der Leader Election in COQ.

Das Theorem zur Zeugeneigenschaft besagt (unter der Annahme der lokalen Prädikate): Es existiert eine Komponente im Netzwerk, die der gewählte Leader einer jeden Komponente ist.

Wir beweisen die Zeugeneigenschaft mithilfe einiger Lemmata, wie zum Beispiel mit dem, in Code-Block 15.3 dargestellten, Lemma.

```
Lemma ancestors_agree:
forall (n:nat)(x y: Component)(prop1: v x) (prop2:v y),
leader(x) = leader (parent_iteration n x).
```

Code-Block 15.3: Hilfslemma für die Zeugeneigenschaft bei der Leader-Election.

Das Lemma drückt aus, dass eine Komponente mit all ihren Vorgängern im Spannbaum (`parent_iteration`) in der Wahl des Leaders übereinstimmt. Wir beweisen das Lemma mit einer Induktion über die Relation `parent`.

15.1.2 Checker

In Code-Block 15.4 ist die Implementierung einer Entscheidungsprozedur des lokalen Prädikats für jede Komponente dargestellt.

```
Definition subchecker (c : checker_input) : bool :=
((( negb (beq c.( leader_i) c.(i))) ∧
beq c.( leader_i) c.(leader_parent_i)) ∧
beq_nat c.(distance_i) ( c.( distance_parent_i)+1) ∧
In_bool c.(parent_i) c.(neighbors)) ∧
forallb_neigbors c.(leader_neighbors) c.(leader_i).
```

Code-Block 15.4: Entscheidungsprozedur des lokalen Prädikats.

Die Implementierung sieht vielleicht zunächst etwas kompliziert aus. Sie setzt jedoch die einzelnen Bedingungen, die für das lokale Prädikat erfüllt sein müssen, direkt um. Die erste Zeile drückt zum Beispiel aus, dass eine Komponente sich nicht selbst als Leader gewählt hat. Wir haben auch hier die Klausel für die Wurzel für eine prägnantere Darstellung weggelassen.

Für die Programmverifikation der Teilchecker müssen wir zeigen, dass der implementierte Teilchecker einer Komponente das lokale Prädikat einer Komponente entscheidet. Der Beweis ist technisch, aber nicht sonderlich kompliziert: er basiert auf syntaktischem Umschreiben in COQ. Entsprechend benötigen wir für die Programmverifikation an dieser Stelle keine zusätzlichen Lemmata, um unsere Argumente zu strukturieren.

15.2 FORMALE INSTANZVERIFIKATION: BIPARTITHEITSTEST

In Abschnitt 15.2.1 stellen wir die Formalisierung des Zeugenprädikats in COQ vor und in Abschnitt 15.2.2 betrachten wir eine Entscheidungsprozedur des Zeugenprädikats für den Checker. Wir betrachten im Rahmen dieser Arbeit nur den Fall der Zertifizierung eines bipartiten Netzwerks in COQ. Der Grund dafür ist, dass der andere Fall eines nicht bipartiten Netzwerks um einiges länger ist.

15.2.1 Zeugenprädikat

In der verwendeten Bibliothek GraphBasics fehlen noch einige Definitionen, die wir für die Fallstudie benötigen. Zum Beispiel gibt es keine Definitionen für bipartite Graphen oder Färbungen eines Graphen.

In Code-Block 15.5 ist unsere Definition einer Bipartition der Komponenten eines Netzwerks in COQ dargestellt.

```
Definition bipartition (arcs: A_set) (color: Component → bool): Prop
    :=
forall (arc : Arc), arc in arcs
    → color (arc_right arc) <> color (arc_left arc).

Definition bipartite (arcs: A_set) :=
exists (color : Component → bool), bipartition arcs color.
```

Code-Block 15.5: Definition einer Bipartition in COQ.

Jeder Komponente wird hierbei eine der Farben `true` oder `false` zugeordnet. Da uns zwei Farben genügen, wählen wir eine Farbe als einen booleschen Wert. Wenn eine Kante aus der Menge der Kanten kommt, die wir betrachten, dann sind ihre Endknoten unterschiedlich gefärbt. Darauf aufbauend definieren wir mit `bipartite` auch, wann ein Netzwerk bipartit ist, wobei wir uns auf die Kantenmenge beziehen. Ein Netzwerk ist bipartit, wenn eine Bipartition der Komponenten existiert.

In Code-Block 15.6 ist das lokale Prädikat für das universell-verteilbare Zeugenprädikat in COQ dargestellt. Das lokale Prädikat `gamma` ist erfüllt für eine Komponenten, wenn die Nachbarschaft der Komponente bipartit ist.

```
Definition gamma (v: V_set)(a: A_set)(g: Connected v a)(v1: Component)
(color: Component → bool) :=
forall (v2 : Component), In v2 v1.neighbors
    → color v1 <> color v2.
```

Code-Block 15.6: Lokales Prädikat für die Zertifizierung eines bipartiten Netzwerks.

In Code-Block 15.7 ist das Theorem der Zeugeneigenschaft in COQ dargestellt.

```
Theorem witness_property:
(forall (v1 : Component), v v1
→ gamma v arcs v1 color) → bipartite arcs.
```

Code-Block 15.7: Zeugen- und Verteilungseigenschaft der Zertifizierung eines bipartiten Netzwerks in COQ.

Die Zeugeneigenschaft ist hier gemeinsam mit der Verteilungseigenschaft angegeben. Wenn das lokale Prädikat für jede Komponente des Netzwerks erfüllt ist, dann ist das Netzwerk bipartit.

15.2.2 Checker

Eine Entscheidungsprozedur für das lokale Prädikat in COQ ist in Code-Block 15.8 dargestellt. Die Liste `nbrs` ist dabei eine Liste, die die Farben der Nachbarn einer Komponente enthält.

```
Fixpoint nbrs_bipartition (col: bool)(nbrs: list component): bool
match nbrs with
    | nil ⇒ true
    | n ::  rest_nbrs ⇒ if (eqb n.col col) then false
       else nbrs_bipartition col rest_nbrs
end.
```

Code-Block 15.8: Entscheidungsprozedur des lokalen Prädikats in COQ.

Für die Programmverifikation des Teilcheckers zeigen wir, dass er das lokale Prädikat für eine Komponente entscheidet.

Teil VI

ENTWURFSMUSTER & WEITERE KLASSEN

Wir stellen in diesem Teil der Arbeit sowohl Entwurfsmuster für die Entwicklung zertifizierender Varianten verteilter Algorithmen vor als auch weitere Methoden zur Übertragung des Konzepts zertifizierender sequentieller Algorithmen auf verteilte Algorithmen.

In Kapitel 16 diskutieren wir Entwurfsmuster. Dabei untersuchen wir zum einen, wie sich bekannte Entwurfsmuster zertifizierender sequentieller Algorithmen übertragen lassen und zum anderen welche speziellen Entwurfsmuster wir für zertifizierende verteilte Algorithmen herausstellen können.

In Kapitel 17 diskutieren wir weitere Klassen zertifizierender verteilter Algorithmen, die sich aus verschiedenen Methoden der Übertragung des Konzepts eines zertifizierenden sequentiellen Algorithmus ergeben. Zum Beispiel diskutieren wir die zusätzliche Zertifizierung von Teilausgaben anstelle lediglich der Zertifizierung der Ausgabe des gesamten Netzwerks.

16 ENTWURFSMUSTER ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

In diesem Kapitel beschäftigen wir uns mit Entwurfsmustern zertifizierender verteilter Algorithmen.

In Abschnitt 16.1 übertragen wir bekannte Entwurfsmuster zertifizierender sequentieller Algorithmen auf die Entwicklung zertifizierender verteilter Algorithmen.

In Abschnitt 16.2 stellen wir spezielle Entwurfsmuster zur Entwicklung zertifizierender verteilter Algorithmen heraus.

In Abschnitt 16.3 zeigen wir auf mit welchen anderen Techniken der Verifikation zertifizierende verteilte Algorithmen kombiniert werden können.

16.1 ÜBERTRAGUNG DER ENTWURFSMUSTER ZERTIFIZIERENDER SEQUENTIELLER ALGORITHMEN

Für zertifizierende sequentielle Algorithmen sind einige Entwurfsmuster bekannt. In [McC+11] werden die folgenden, für uns interessanten, allgemeinen Muster zum Entwurf einer zertifizierenden Variante eines sequentiellen Algorithmus aufgeführt:

- charakterisierende Theoreme (siehe Abschnitt 16.1.1),
- Komposition (siehe Abschnitt 16.1.2) und
- Reduktion (siehe Abschnitt 16.1.3).

Die meisten dieser Entwurfsmuster lassen sich direkt auf zertifizierende verteilte Algorithmen übertragen. Wir gehen deswegen nur in Kürze auf die jeweiligen Entwurfsmuster in den entsprechenden Abschnitten ein und verweisen für mehr Details auf [McC+11].

16.1.1 Charakterisierende Theoreme

Durch ein charakterisierendes Theorem lässt sich häufig eine Idee für die Zertifizierung ableiten. In [McC+11] wird dabei auch folgendes Muster beobachtet:

> Interestingly these characterizations follow a certain pattern: One direction of the proof of the characterization is easy, and this side corresponds to required simplicity of the witness. The more difficult direction is the one required to establish the existence of a witness.

Das heißt der Beweis der Zeugeneigenschaft eines Zeugenprädikats ist meist einfach, während der Beweis zur Vollständigkeit des Zeugenprädikats schwieriger ist.

Wir haben zum Beispiel eine Charakterisierung der Distanzfunktion für eine zertifizierende Variante der Berechnung kürzester Pfade in einem Netzwerk benutzt (siehe Fallstudie in Abschnitt 9.2).

Da verteilte Algorithmen so entworfen werden, dass sie auf einem Netzwerk ausgeführt werden, spielen Theoreme aus der Graphentheorie eine wichtige Rolle. Zum einen gibt es verteilte Graphalgorithmen, bei denen das Netzwerk selbst der Eingabegraph ist [Erc13], wie zum Beispiel bei dem verteilten Bipartitheitstest. Für dessen Zertifizierung haben wir eine Charakterisierung für nicht bipartite Graphen genutzt (siehe Fallstudie in Abschnitt 9.8).

Zum anderen gibt es auch verteilte Algorithmen, die keine verteilten Graphalgorithmen sind, für deren Zertifizierung dennoch graphentheoretische Erkenntnisse interessant sind. Beispielsweise haben wir bei der Leader-Election einen Spannbaum im Netzwerk für die Zertifizierung genutzt (siehe Fallstudie in Abschnitt 9.7). Für den Spannbaum wiederum haben wir eine graphentheoretische Charakterisierung genutzt.

16.1.2 Komposition

Bei dem Entwurfsmuster der Komposition geht es darum, eine zertifizierende Variante für einen Algorithmus zu entwickeln, indem zwei bereits zertifizierende Algorithmen komponiert werden. In unseren Fallstudien in Kapitel 9 haben wir keine Komposition vorgestellt. Wir haben jedoch eine zertifizierende Komposition einer Konsensfindung mit einem Bipartitheitstest in Abschnitt 7.3 in Beispiel 23 skizziert.

Wichtig für die Übertragung der Komposition auf zertifizierende verteilte Algorithmen ist die von uns gewählte Definition eines verteilbaren Zeugenprädikats. Die Komposition gelingt, da die Konjunktion zweier verteilbarer Prädikate wieder ein verteilbares Prädikat ist.

Bei der Komposition zweier zertifizierender sequentieller Algorithmen wird diese Konjunktion umgesetzt, indem erst das eine Zeugenprädikat und nach dessen positiver Entscheidung das andere Zeugenprädikat entschieden wird.

Der neue Zeuge besteht aus den beiden berechneten Zeugen der komponierten Algorithmen. Die Konsistenz wird genauso für beide Zeugen nacheinander geprüft, womit die Konsistenz des neuen Zeugen eine Konjunktion der Konsistenz beider Zeugen ist.

16.1.3 Reduktion

Reduzieren wir ein Problem P auf ein anderes Problem P′, für dessen Lösung es bereits einen zertifizierenden Algorithmus gibt, so gewinnen wir einen zertifizierenden Algorithmus für das Problem P. Wir können das Entwurfsmuster der Reduktion genauso für zertifizierende verteilte Algorithmen übernehmen.

Wir haben zum Beispiel die Breitensuche auf das Problem der Berechnung kürzester Pfade reduziert und somit eine zertifizierende Breitensuche in Netzwerken entwickelt (siehe Fallstudie in Abschnitt 9.4).

16.2 SPEZIELLE ENTWURFSMUSTER ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Wir betrachten spezielle spezielle Entwurfsmuster zertifizierender verteilter Algorithmen. Durch unsere Fallstudien (siehe Kapitel 9) können wir die folgenden Entwurfsmuster herausstellen:

- Nutzung eines Spannbaums (siehe Abschnitt 16.2.1),
- Nutzung einer Distanzfunktion (siehe Abschnitt 16.2.1),
- Nutzung einer Einigkeit (siehe Abschnitt 16.2.2),
- Nutzung der optimalen Substruktur des zu lösenden Problems (siehe Abschnitt 16.2.3),
- Nutzung eines zertifizierenden sequentiellen Algorithmus (siehe Abschnitt 16.2.4) und
- Nutzung eines selbst-stabilisierenden Algorithmus (siehe Abschnitt 16.2.5).

16.2.1 Spannbaum und Distanz

In unseren Fallstudien haben wir des Öfteren einen Spannbaum als Zeugen benutzt. Beispielsweise bei der Zertifizierung der Leader-Election oder dem Bipartitheitstest. Dabei haben wir den Spannbaum als Argument für die Existenz benutzt; zum Beispiel die Existenz einer Komponente in einem Netzwerk bei der Leader-Election oder der Existenz eines Nachrichtenkanals beim Bipartitheitstest.

Für die Zertifizierung eines Spannbaum wiederum haben wir eine Charakterisierung mithilfe der Distanz in einem Spannbaum gewählt. Dabei ist die Korrektheit der Distanzfunktion elegant mit einem lokalen Prädikat verteilbar. Die Distanz ermöglicht es uns, Induktionsbeweise für Eigenschaften in einem Netzwerk zu führen.

16.2.2 Einigkeit

In den Fallstudien war häufig ein Prädikat zur Einigkeit eines der Verteilungsprädikate des verteilbaren Zeugenprädikats. Zum Beispiel bei der Zertifizierung eines Spannbaums die Einigkeit über die Identität der Wurzel des Spannbaums (siehe Abschnitt 9.3), bei der Zertifizierung des Broadcast die Einigkeit über eine Nachricht (siehe Abschnitt 9.6) oder bei der Konsensfindung eine Einigkeit über eine Wahl (siehe Abschnitt 9.5).

Die Einigkeit in einem Netzwerk kann dabei auf die Einigkeit in allen Nachbarschaften herunter gebrochen werden und ist damit elegant durch ein lokales Prädikat universell-verteilbar. Wir nutzen Einigkeit als ein Argument, um lokal auf eine Eigenschaft global zu schließen.

16.2.3 Optimale Substruktur

Eine optimale Substruktur ist interessant für Optimierungsprobleme. Ein Problem hat optimale Substruktur, wenn wir eine optimale Lösung des Problems aus optimalen Lösungen von Teilproblemen des Problems schließen können. Probleme mit optimaler Substruktur bringen schon eine Verteilung mit sich, da optimale Lösungen für die Teilprobleme berechnet werden können. Für sequentielle Algorithmen, die Probleme mit optimaler Substruktur lösen, lassen sich deswegen auch besonders einfach verteilte Varianten finden [Ber82; Ray13].

Wir haben mit dem Problem der Berechnung kürzester Pfade ein Problem mit optimaler Substruktur gesehen (siehe Abschnitt 9.2). In diesem Fall haben wir die zertifizierende Variante aus einem zertifizierenden sequentiellen Algorithmus für das gleiche Problem abgeleitet. Dabei war die Verteilungseigenschaft jedoch bereits implizit

im Zeugenprädikat enthalten und der Grund dafür ist die optimale Substruktur des Problems.

16.2.4 Zertifizierender sequentieller Algorithmus

Ein Entwurfsmuster für einen zertifizierenden verteilten Algorithmus ist es, von einem zertifizierenden Algorithmus auszugehen, der das gleiche Problem sequentiell löst.

In den Fallstudien haben wir das anhand der Berechnung kürzester Pfade, sowie anhand des Bipartitheitstest gesehen. Die Zeugeneigenschaft konnten wir dabei jeweils übernehmen. Die Herausforderung besteht dann darin, eine sinnvolle Verteilungseigenschaft zu finden.

16.2.5 Selbst-stabilisierender Algorithmus

Wir haben bisher in dieser Arbeit kein Beispiel für einen zertifizierenden verteilten Algorithmus gesehen, den wir von einem selbst-stabilisierenden Algorithmus ausgehend entwickelt haben. Wir zeigen jedoch im folgenden Kapitel 17 einen selbst-stabilisierenden Algorithmus, der auch ein zertifizierender verteilter Algorithmus ist.

Für einen selbst-stabilisierenden Algorithmus ist garantiert, dass irgendwann ein korrekter Zustand im Netzwerk erreicht wird. Für die Selbst-Stabilisierung wird jedoch von einem anderen Fehlermodell ausgegangen (siehe Abschnitt 5.2 für unser Fehlermodell). Vereinfacht und intuitiv gesagt sieht das Fehlermodell der Selbst-Stabilisierung wie folgt aus: endlich lange können diverse Fehler passieren, aber nach endlich langer Zeit ist garantiert, dass das verteilte System fehlerfrei ist. Dabei ist für uns vor allem interessant, dass der Teilalgorithmus einer Komponente bei der Selbst-Stabilisierung als prinzipiell vertrauenswürdig gesehen wird im Gegensatz zu unserem Fehlermodell.

Eine einfache Art der Selbst-Stabilisierung ist deswegen die wiederholte Ausführung für einen terminierenden verteilten Algorithmus. Da der Algorithmus selbst als vertrauenswürdig gilt und das System irgendwann fehlerfrei ist, wird damit garantiert, dass irgendwann ein korrekter Zustand im System erreicht wird. Viele selbst-stabilisierenden Algorithmen sind aufgrund dieses anderen Fehlermodells nicht zertifizierend.

Dennoch gibt es auch Gemeinsamkeiten, die bei der Entwicklung eines zertifizierenden verteilten Algorithmus helfen können. Ein selbst-stabilisierender Algorithmus stabilisiert sich selbst zu einem korrekten Zustand hin. Häufig hat ein selbst-stabilisierender Algortihmus dafür einen Mechanismus, der entscheidet, wie der Zustand repariert wer-

den muss, um korrekt zu sein. Manchmal kann dieser Mechanismus auch benutzt werden, um zu bezeugen, dass der Zustand korrekt ist.

Wir stellen im folgenden Kapitel in Abschnitt 17.2 einen zertifizierenden verteilten Algorithmus vor, den wir ausgehend von einem selbststablisierenden Algorithmus entwickelt haben.

16.3 KOMBINATION MIT ANDEREN TECHNIKEN

In diesem Abschnitt diskutieren wir Möglichkeiten zertifizierende verteilte Algorithmen mit anderen Techniken zu kombinieren. In [McC+11] wird vorgeschlagen, wie für sequentielle Algorithmen Zertifizierung und Verifikation fernab der formalen Instanzverifikation kombiniert werden können.

In Abschnitt 16.3.1 schlagen wir vor, wie für verteilte Algorithmen Zertifizierung und Verifikation kombiniert werden können. In Abschnitt 16.3.2 weisen wir außerdem auf eine Simulationsumgebung für zertifizierende verteilte Algorithmen hin. Die Simulationsumgebung wurde im Rahmen einer Bachelorarbeit entwickelt, die von der Autorin dieser Arbeit betreut wurde [Aki15].

16.3.1 Techniken der Verifikation

In [McC+11] wird für sequentielle Algorithmen vorgeschlagen einige Eigenschaften der Korrektheit zur Laufzeit zu zertifizieren und andere zu verifizieren. Als Beispiel wird ein Algorithmus angeführt, der eine Sequenz sortiert. Dass die Ausgabesequenz sortiert ist, wird zur Laufzeit verifiziert. Dass die Ausgabesequenz aber eine Permutation der Eingabesequenz ist, wird durch eine Programmverifikation sichergestellt: der implementierte Algorithmus nutzt eine verifizierte Funktion, die lediglich die Werte an zwei Positionen der Sequenz tauscht. Die Idee dabei ist, dass einige Eigenschaften einfacher zu zertifizieren sind, während andere einfacher zu verifizieren sind.

16.3.1.1 *Lebendigkeitseigenschaften*

Für verteilte Algorithmen, die nicht terminieren, müssen häufig sowohl Sicherheitseigenschaften als auch Lebendigkeitseigenschaften für eine korrekte Ausführung gezeigt werden. Das ist zum Beispiel bei Mutex-Algorithmen der Fall. Durch Laufzeitverifikation können wir Lebendigkeitseigenschaften nicht zeigen, da Lebendigkeitseigenschaften über unendlichen Abläufen definiert sind. Wir können also für

jeden endlichen Präfix eines Ablaufs nicht entscheiden, ob der Ablauf im Sinne der Lebendigkeitseigenschaft korrekt ist [Ray+11].

Eine Idee für verteilte Algorithmen könnte es deswegen sein, Lebendigkeitseigenschaften zu verifizieren und Sicherheitseigenschaften zu zertifizieren. Im folgenden Kapitel stellen wir einen zertifizierenden Mutex-Algorithmus vor, für den wir eine Sicherheitseigenschaft zertifizieren.

16.3.1.2 *Fehlertoleranz*

In unserem Framework haben wir außerdem schon eine Kombination von Zertifizierung und Verifikation auch fernab der formalen Instanzverifikation beschrieben. VERDI stellt einige verifizierte Algorithmen zur Fehlertoleranz bereit, die wir für unsere Checker zusätzlich benutzen können, um diese fehlertolerant zu gestalten.

16.3.1.3 *Kommunikation & zeitliche Aspekte*

In [AL19] haben Akili und Lorenz auf der industriellen Fallstudie aus Kapitel 11 aufgebaut, indem sie die zertifizierende verteilte Auktion mit zwei weiteren Techniken der Laufzeitverifikation kombiniert haben. Zum einen haben sie eine Technik, die auf Session-Types aufbaut genutzt, um Eigenschaften über die Kommunikation zu verifizieren, wie zum Beispiel, dass eine Nachricht irgendwann einmal tatsächlich verschickt wurde. Zum anderen haben sie einen Ansatz des Monitoring gewählt, mit dem sie die Einhaltung von Deadlines während der Ausführung verifizieren.

In der industriellen Fallstudie hat jeder Roboter zur Teilnahme an der Auktion sein eigenes Gebot zunächst mit einem sequentiellen Algorithmus berechnet. An dieser Stelle wäre auch eine Kombination mit einem zertifizierenden sequentiellen Algorithmus denkbar.

16.3.1.4 *Ausführungskorrektheit*

Weiterhin könnten Techniken zur *Ausführungskorrektheit* interessant sein. Gemeint ist damit das Szenario, dass ein Programm für eine Eingabe auf irgendeiner Komponente eines verteilten Systems ausgeführt wird und dort eine Ausgabe berechnet. Dadurch gibt es ein Interesse daran, dass die Ausgabe tatsächlich mit dem entsprechenden Programm, sowie der Eingabe berechnet wurde.

Es gibt einige Techniken, die die Korrektheit der Ausführung sicherstellen [GGP10; KH13; ZPK14; WB15]. Im Kontext eines zertifizierenden verteilten Algorithmus könnten diese Techniken interessant sein, um sicherzustellen, dass tatsächlich ein verifizierter Teilchecker ausgeführt wurde.

16.3.2 Simulationsumgebung

In [Aki15] wird eine Simulationsumgebung für zertifizierende verteilte Algorithmen beschrieben. Dabei wird die, in Abschnitt 10.1 beschriebene, verteilte Architektur für Checker umgesetzt. Jeder Teilchecker ist eine logische Einheit auf einer Komponente des Netzwerks. Die Kommunikation einer Komponente mit ihrem Teilchecker, sowie zwischen den Teilcheckern benachbarter Komponenten erfolgt dabei, wie in dem Abschnitt 10.1 beschrieben.

Die Simulationsumgebung baut auf dem weit verbreiteten Simulator OMNeT++ auf [Var10]. Als Begründung für diese Wahl wird in [Aki15] neben der weiten Verbreitung und guten Dokumentation auch eine ansprechende Visualisierung, die kostenfreie Verfügbarkeit, sowie die leichte Bedienbarkeit aufgeführt. Die Simulationsumgebung wird anhand der Fallstudie der zertifizierenden verteilten Berechnung kürzester Pfade illustriert.

Für den Entwurf eines zertifizierenden verteilten Algorithmus ist diese Simulationsumgebung interessant, da er so unter realistischen Netzwerkbedingungen analysiert werden kann. Dabei kann zum Beispiel der Effekt von Time-Outs, aber auch diversen Fehlermodellen untersucht werden.

17 WEITERE KLASSEN ZERTIFIZIERENDER VERTEILTER ALGORITHMEN

Wir haben in dieser Arbeit eine Methode vorgestellt, das Konzept zertifizierender sequentieller Algorithmen auf verteilte Algorithmen zu übertragen und damit eine Klasse zertifizierender verteilter Algorithmen definiert. In diesem Kapitel diskutieren wir andere Klassen zertifizierender verteilter Algorithmen.

In Abschnitt 17.1 stellen wir die Idee vor, auch Teilausgaben und nicht nur die gesamte Ausgabe in einem Netzwerks zu zertifizieren. Wir illustrieren diese Idee an einer Fallstudie, die wir in [Völ19] veröffentlicht haben.

In Abschnitt 17.2 zeigen wir auf, wie die Zertifizierung eines verteilten Algorithmus aussehen kann, der nicht terminiert.

17.1 LOKALE KORREKTHEIT

In Abschnitt 17.1.1 formulieren wir zunächst das Problem der lokalen Korrektheit. In Abschnitt 17.1.2 stellen wir dann eine Fallstudie basierend auf der Berechnung kürzester Pfade vor. In Abschnitt 17.1.3 schließen wir unsere Betrachtungen mit einer Diskussion ab.

17.1.1 Problem der lokalen Korrektheit

Unter lokaler Korrektheit verstehen wir, dass eine Teilausgabe einer Komponente für eine Teileingabe der Komponente korrekt ist. Wir motivieren zunächst das Problem der lokalen Korrektheit.

17.1.1.1 *Motivation*

Lokalität ist im Allgemeinen ein wichtiger Aspekt verteilter Algorithmen. So beschreibt Peleg in [Peloo], dass viele Aufgaben verteilter Algorithmen lokal erledigt werden können und nennt das Phänomen *locality of reference*:

> [...] "locality of reference", namely, relying on the fact that performing these tasks requires [components] to know

> more about their immediate neighborhood and less about the rest of the world. Furthermore, certain tasks are local in nature and only involve [components] located in a small region in the network.

Während lokale Korrektheit nicht für alle Probleme, die verteilte Algorithmen lösen, sinnvoll definiert werden kann, denken wir, dass es einige Probleme gibt – die wie Peleg sagt lokaler Natur sind – für die sich eine solche Betrachtung anbietet.

17.1.1.2 *Szenario für die lokale Korrektheit*

Interessant ist die Frage nach der lokalen Korrektheit, wenn wir uns das folgende Szenario vorstellen. Angenommen wir haben für ein Problem einen zertifizierenden verteilten Algorithmus gegeben, der dieses Problem löst. Dann wissen wir, dass wenn der zugehörige Checker für ein Eingabe-Ausgabe-Paar und einen Zeugen akzeptiert, dass dann das Eingabe-Ausgabe-Paar korrekt ist.

Nehmen wir nun aber an, dass eine einzige Komponente fehlerhaft gerechnet hat und entsprechend eine falsche Teilausgabe besitzt. Konsequenterweise akzeptiert der Checker in diesem Fall nicht und die Laufzeitverifikation schlägt fehl. Eine Lösung könnte nun sein, die Berechnung der gesamten Ausgabe zu wiederholen oder auch einen anderen verteilten Algorithmus zur Lösung des Problems zu nutzen.

Wir beobachten jedoch an dieser Stelle, dass für die Teilausgaben der anderen Komponenten weiterhin gelten könnte, dass sie korrekt sind. Einige Komponenten waren durch den Fehler eventuell beeinflusst und haben somit wahrscheinlich eine falsche Teilausgabe berechnet. Andere Komponenten wiederum sind eventuell von dem Fehler gar nicht betroffen und haben eine korrekte Teilausgabe berechnet.

Dieses Szenario motiviert das Problem der lokalen Korrektheit. Es könnte für einige Anwendungsfälle interessant sein, möglichst viele korrekte Teilausgaben zu identifizieren, sodass selbst bei einer wiederholten Berechnung auf diese Ergebnisse zurückgegriffen werden kann.

17.1.1.3 *Problembeschreibung*

Bei dem Problem der lokalen Korrektheit wird somit die Frage behandelt, ob das Paar aus Teileingabe und Teilausgabe einer Komponente korrekt ist bezüglich einer Eingabe-Ausgabe-Spezifikation. Um die Frage der lokalen Korrektheit sinnvoll stellen zu können, benötigen wir eine Definition dafür, was Korrektheit für ein einzelnes Paar bestehend aus Teileingabe und Teilausgabe bedeutet.

Das schließt einige Probleme aus. Beispielsweise fordert die Spezifikation des Problems der Leader-Election, dass die Ausgabe eine Komponente des Netzwerks als eindeutigen Leader bestimmt. Wenn eine Komponente sich selbst als Leader gewählt hat, dann kann das in Abhängigkeit der Teilausgaben aller anderen Komponenten korrekt oder inkorrekt sein. Deshalb bietet sich diese Spezifikation offensichtlich nicht für die Fragestellung des Problems der lokalen Korrektheit an. Anders sieht es jedoch bei dem Problem der Berechnung kürzester Pfade aus, das wir im folgenden Abschnitt betrachten.

17.1.2 Fallstudie: Kürzeste Pfade

Ausgangspunkt für unsere Fallstudie ist die zertifizierende Variante der verteilten Berechnung kürzester Pfade in einem Netzwerk, wie wir sie in Abschnitt 9.2 vorgestellt haben. Wir untersuchen im Folgenden, inwiefern ein Zeuge oder der Checker hilfreich sein können, um die lokale Korrektheit für einige Komponenten des Netzwerks zu entscheiden. Die berechnete Teilausgabe einer Komponente ist korrekt, wenn es sich dabei um die Distanz der Komponenten zur Quelle s handelt. Dabei ist die Funktion d wieder die Distanzfunktion bezüglich s, die Funktion D die berechnete Funktion im Netzwerk und die Funktion c beschreibt die Kosten der Kanäle des Netzwerks.

Für unsere Betrachtungen nehmen wir an, dass eine Komponente $v \neq s$ ihre Teilausgabe $D(v)$ fehlerhaft berechnet, wobei der Typ korrekt ist, also eine positive reelle Zahl ist. Es gilt also $D(v) \neq d(v)$, $D(v) \geqslant 0$ und $v \neq s$.

Jede andere Komponente $u \neq v$ rechnet korrekt und berechnet damit ihre Teilausgabe $D(u)$ entsprechend folgerichtig zur fehlerhaften Teilausgabe $D(v)$. Folgerichtiges Rechnen bedeutet in diesem Fall, dass für jede Komponente $u \neq v$ gilt:

$$D(u) = 0 \text{ falls } u \text{ die Quelle ist} \tag{17.1}$$

$$D(u) \leqslant D(w) + c(u,w) \text{ für alle Nachbarn } w \tag{17.2}$$

$$D(u) = D(w) + c(u,w) \text{ für mindestens einen Nachbarn } w \tag{17.3}$$

Daraus folgt, dass folgerichtiges Rechnen bedeutet, dass für alle Komponenten mit Ausnahme der fehlerhaft rechnenden Komponente die Eigenschaften der Charakterisierung der Distanzfunktion gelten. Wäre dies nicht der Fall, zum Beispiel für die Eigenschaft (17.2), würde es aus der Perspektive der Komponente u einen kürzeren Pfad geben, den sie nicht gewählt hat. Das steht im Widerspruch dazu, dass die Komponente u korrekt rechnet.

Die berechnete Distanz entspricht allerdings nur der Distanz, wenn die Eigenschaften der Charakterisierung für alle Komponenten gilt. Da die Teilausgabe von v nicht korrekt ist, gilt für die Teilausgabe

jeder anderen Komponente u also, dass sie korrekt ($D(u) = d(u)$) oder inkorrekt ($D(u) \neq d(u)$) sein kann.

17.1.2.1 *Identifikation der fehlerhaften Komponente*

Falls der Checker nicht akzeptiert, lehnt mindestens ein Teilchecker ab, da das Zeugenprädikat universell-verteilbar ist. Da ein Teilchecker das lokale Prädikat für seine Komponente genau dann positiv entscheidet, wenn in deren Nachbarschaft die Eigenschaften der Charakterisierung gelten, lehnt einzig der Teilchecker der fehlerhaften Komponente v ab. Dadurch wird die fehlerhafte Komponente v durch das Verhalten ihres Teilcheckers identifiziert.

Wir unterscheiden die beiden Fälle, wie der Teilchecker von v ablehnt. Wenn die Dreiecksungleichung (17.2) nicht gilt für v, dann hat v seine Distanz zu groß berechnet – also $D(v) > d(v)$. Dann existiert ein Nachbar w von v, sodass $d(v) = d(w) + c(v, w)$ und folglich $D(v) > d(w) + c(v, w)$. Wenn die Gleichung (17.3) nicht gilt, dann hat v seine Distanz wiederum zu klein berechnet $D(v) < d(v)$.

Ein Teilchecker unterscheidet die beiden Fälle, indem er erst die Eigenschaft (17.2) prüft und nur wenn diese erfüllt ist, noch die Eigenschaft (17.3) prüft. Der Teilchecker der Komponente v kann somit entscheiden, in welche Richtung die Komponente v fehlerhaft gerechnet hat.

17.1.2.2 *Fortpflanzung des Fehlers*

Wir analysieren nun, wie sich der Fehler von v im Netzwerk bei der Berechnung der Distanzen der anderen Komponenten fortpflanzt. Dafür betrachten wir Teilnetzwerke des Netzwerks. Zur Illustration unserer Argumente nehmen wir dafür das Netzwerk N und die Teilnetzwerke I-VI in Abbildung 17.1 an.

Wir argumentieren unter welchen Umständen eine Komponente eine korrekte Teilausgabe berechnet hat. Dabei bedienen wir uns an Argumenten, die die identifizierte Fallunterscheidung des Teilcheckers von v nutzt, sowie Argumenten über die berechnete Elternrelation, die der berechnete Zeuge ist.

Wir argumentieren aber auch darüber, wie sich eine mögliche Topologie eines Netzwerks auswirkt. Wir nehmen zwar bei Netzwerken keine spezielle Topologie an, wenn es jedoch beispielsweise einen Gelenkpunkt geben sollte, dann lässt diese Topologie mehr Schlüsse für die lokale Korrektheit zu. Ein *Gelenkpunkt* in einem Graph ist ein Knoten, dessen Entfernung zu zwei Zusammenhangskomponenten führt. In der Abbildung 17.1 ist x ein Gelenkpunkt.

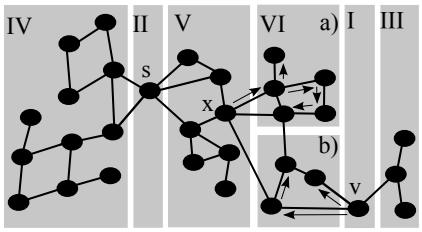


Abbildung 17.1: Netzwerk zur Illustration der Fehlerfortsetzung in Teilnetzwerke. Die Nummerierung der Teilnetzwerke entspricht dabei der Reihenfolgen, in der wir über sie argumentieren. Die Kosten der Kanäle des Netzwerks sind nicht dargestellt. Die berechnete Elternrelation (Zeuge) ist nur für das Teilnetzwerk VI dargestellt. x ist ein Gelenkpunkt. Die Abbildung ist unserer Veröffentlichung der Fallstudie entnommen [Völ19].

17.1.2.3 *Teilnetzwerk I und II*

Das Teilnetzwerk I enthält ausschließlich die fehlerhafte Komponente v, für deren Teilausgabe wir wissen, dass sie inkorrekt ist. Außerdem wissen wir, dass v nicht die Quelle s ist – die einzige Komponente des Teilnetzwerks II. Für die Quelle gilt, da sie fehlerfrei rechnet, dass ihre Teilausgabe $D(s) = 0$ korrekt ist.

17.1.2.4 *Teilnetzwerk III*

Für jede Komponente u des Teilnetzwerks III gilt, dass ihre Teilausgabe $D(u)$ inkorrekt ist. Der Grund ist, dass jeder Pfad von s nach u über v führt. Genauer gesagt, u berechnet ihre Distanz zu v korrekt. Wir bezeichnen die Distanz von u zu v als $d_v(u)$. Folglich gilt $D(u) = d_v(u) + D(v)$. Wenn die Teilausgabe $D(u)$ korrekt wäre, dann würde $D(u) = d_v(u) + d(v)$ gelten und damit $D(v) = d(v)$. Das ist allerdings ein Widerspruch zu unserer Annahme $D(v) \neq d(v)$. Es gilt also $D(u) \neq d(u)$ für alle Komponenten u des Teilnetzwerks III.

17.1.2.5 *Teilnetzwerk IV*

Für jede Komponente u des Teilnetzwerks IV gilt, dass ihre Teilausgabe $D(u)$ korrekt ist. Die Berechnung der Teilausgabe $D(u)$ kann nur dann durch die fehlerhafte Teilausgabe $D(v)$ beeinflusst werden, wenn ein (vermeintlich) kürzester Pfad von u zu s über v führen würde. Solch ein s-u Pfad würde jedoch s mindestens zweimal enthalten. Da die Kosten aller Kanäle jeweils positiv sind und auch die fehlerhafte Teilausgabe $D(v)$ positiv ist, kann eine folgerichtig rechnende Komponente solch einen Pfad nicht als kürzesten Pfad berechnen. Der Fehler pflanzt sich also nicht in das Teilnetzwerk IV fort.

17.1.2.6 *Teilnetzwerk V*

Für jede Komponente u des Teilnetzwerks V gilt, dass ihre Teilausgabe $D(u)$ korrekt ist, falls sich die fehlerhafte Komponente v nach oben verrechnet hat: $D(v) > d(v)$. Es gilt schließlich $d(v) = d_x(v) + d(x)$ und daraus folgt $D(v) > d_x(v) + d(x)$. Wegen der positiven Kosten der Kanäle gilt außerdem für jede Komponente u, dass $d(u) \leqslant d(x)$.

17.1.2.7 *Teilnetzwerk VI*

Für das Teilnetzwerk VI betrachten wir zusätzlich den berechneten Zeugen – die Elternrelation, die einen Kürzeste-Pfade-Baum induziert. Wir benutzen die Elternrelation, um für die Komponenten zu unterscheiden, ob sie Nachfolger von v sind oder nicht.

17.1.2.8 *Teilnetzwerk VI a*

Das Teilnetzwerk VI a enthält nur Komponenten, die keine Nachfolger von v sind. Falls die Komponente v ihre Distanz zu klein berechnet hat ($D(v) < d(v)$), dann haben alle Komponenten des Teilnetzwerks VI a eine korrekte Teilausgabe. Der Grund dafür ist, dass diese Komponenten v nicht als ihren Vorgänger gewählt haben, obwohl die Komponente v behauptet hat, eine kleinere Distanz zu haben als sie eigentlich hat.

17.1.2.9 *Teilnetzwerk VI b*

Das Teilnetzwerk VI b hingegen enthält nur Komponenten, die Nachfolger von v sind. Eine Komponente u des Teilnetzwerks hat deswegen die folgende Teilausgabe berechnet $D(u) = d_v(u) + D(v)$. Damit sind die Teilausgaben des Teilnetzwerks potenziell falsch. Im Gegensatz zu den Teilausgaben der Komponenten des Teilnetzwerks III, könnten sie aber auch immer noch korrekt sein. Es könnte schließlich einen s-u Pfad geben, der v nicht enthält und dessen summierte Kosten genau $D(u)$ sind.

Wir wissen außerdem, dass die Elternrelation in dem Teilnetzwerk korrekt berechnet ist, falls die Komponente v zu groß berechnet hat, da die Komponenten des Teilnetzwerks v dennoch aus Mangel an Alternativen als Vorgänger gewählt haben.

17.1.2.10 *Gelenkpunkte und Fehlerfortpflanzung*

Wir haben für das Netzwerk aus Abbildung 17.1 einige Schlüsse auf Basis der Topologie gefolgert, denn die Komponenten s, v und x sind Gelenkpunkte. Für die Fehlerfortpflanzung haben sich Gelenkpunkte

in unserer Fallstudie als besonders interessant erwiesen. Alle Argument, die sich dabei auf die Komponente x beziehen, funktionieren für jede Komponente, deren Entfernung zu mindestens zwei Zusammenhangskomponenten führen würde, sodass die fehlerhafte Komponente und die Quelle in verschieden Zusammenhangskomponenten sind.

Es gibt auch verteilte Algorithmen, um Gelenkpunkte in Netzwerken zu identifizieren [XL10]. Da wir von statischen Netzwerken ausgehen, könnten sie aber auch zur Initialisierung bekannt sein.

17.1.2.11 *Beliebige Topologie*

Einige Netzwerke enthalten keine Gelenkpunkte. Alle Argumente, die wir auf Basis der berechneten Elternrelation erbracht haben, sind unabhängig von der Topologie. Darüber hinaus ist auch die Identifizierung der fehlerhaften Komponente durch den Checker, sowie dessen Fallunterscheidung in welche Richtung sich die Komponente verrechnet hat unabhängig von der Topologie. Der berechnete Zeuge und der Checker können also dabei helfen die lokale Korrektheit für zumindest einige Komponenten zu entscheiden.

17.1.3 Diskussion

Für die Betrachtung der lokalen Korrektheit benötigen wir eine Spezifikation dafür, was es bedeutet, dass eine Komponente eine korrekte Teilausgabe berechnet hat. Für die Fallstudie der kürzesten Pfade ergibt sich diese Spezifikation direkt aus der Spezifikation des Problems, vor allem weil das Problem eine optimale Substruktur aufweist (siehe Abschnitt 16.2).

17.1.3.1 *Übertragung auf andere Probleme*

Für andere Probleme ist zunächst oft nicht klar, was die Korrektheit einzelner Teilausgaben bedeuten soll. Wie bereits diskutiert, ist das zum Beispiel bei dem Problem der Leader-Election nicht klar.

Hier könnte es interessant sein, das Problem der lokalen Korrektheit zu erweitern. Statt einzelne korrekte Teilausgaben zu betrachten, könnte es interessant sein ganze Teilnetzwerke zu betrachten. Im Falle der Leader-Election könnte die Korrektheit in einem Teilnetzwerk dann bedeuten, dass in einem Teilnetzwerk (bestimmter Größe) einstimmig ein Leader gewählt wurde.

17.1.3.2 *Erweiterung für lokale Zertifizierung*

Wir haben aufgezeigt, dass der berechnete Zeuge und der Checker in unserer Fallstudie dabei helfen die lokale Korrektheit für zumindest einige Komponenten zu entscheiden. Dabei sind wir von einer Zertifizierung ausgegangen, die wir so entwickelt haben, dass die Ausgabe im Netzwerk zertifiziert wird. Es könnte deswegen interessant sein eine zertifizierende Variante eines verteilten Algorithmus mit Hinblick auf die lokale Korrektheit zu entwerfen.

17.2 ZERTIFIZIERUNG & NICHT-TERMINIERUNG

In Abschnitt 17.2.1 stellen wir eine zertifizierende Variante eines Mutex-Algorithmus vor. Dabei gestalten wir ihn so, dass er zertifiziert, dass immer nur eine Komponente Zugriff auf eine Ressource hat. In Abschnitt 17.2.2 erläutern wir einige allgemeine Ansätze zur Zertifizierung verteilter Algorithmen, die nicht terminieren.

17.2.1 Fallstudie: Mutex-Algorithmus

Wir spezifizieren zunächst das Problem, das ein Mutex-Algorithmus löst.

17.2.1.1 *Spezifikation des Problems*

Das Problem des Mutex oder auch wechselseitigen Ausschluss besteht darin, den exklusiven Zugriff zum Beispiel auf eine Ressource zu regeln. Der exklusive Zugriff ist dabei durch die folgenden Eigenschaften spezifiziert [Rei10]:

- Jede Komponente, die Zugriff möchte, erhält ihn irgendwann nach endlicher Zeit.
- Jederzeit hat höchstens eine Komponente Zugriff.

Die erste Eigenschaft ist eine Lebendigkeitseigenschaft, die wir mit zertifizierenden Algorithmen nicht verifizieren können beziehungsweise nur durch eine Reduktion der Zertifizierung auf vollständige Verifikation. An einer Zertifizierung bei der die Zeugeneigenschaft die Korrektheit des Algorithmus voraussetzt, sind wir nicht interessiert. Wir betrachten für die Zertifizierung deswegen nur die zweite Eigenschaft, eine Sicherheitseigenschaft.

17.2.1.2 *Algorithmus*

Einige Mutex-Algorithmen funktionieren so, dass ein Token über einen Spannbaum im Netzwerk weitergereicht wird und der Besitz des Token einer Komponente den Zugriff gewährt [Ray13]. Die genaue Funktionsweise eines solchen Algorithmus ist für uns an dieser Stelle uninteressant. Wir zertifizieren eine strengere Eigenschaft und zwar, dass nicht höchstens, sondern genau eine Komponente im Netzwerk das Token besitzt.

Varghese beschreibt in seiner Doktorarbeit einen selbst-stabilisierenden Mutex-Algorithmus, bei dem ein Token in einem Spannbaum weitergereicht wird [Var93]. Dieser Algorithmus besitzt einen Mechanismus für die Selbst-Stabilisierung, bei dem durch lokale Reparaturen des Netzwerkzustands ein korrekter Zustand im Sinne der Sicherheitseigenschaft erreicht wird. Wir wandeln den Mechanismus für unsere Zwecke so ab, dass der Algorithmus die geforderte Sicherheitseigenschaft zertifiziert.

17.2.1.3 *Zertifizierung*

Der Spannbaum über den das Token weitergereicht wird, kann zertifizierend konstruiert werden, wie in Abschnitt 9.3 beschrieben. Dabei konstruieren wir ihn so, dass anfangs die Wurzel das Token besitzt. Für die Zertifizierung kommt hinzu, dass der Teilchecker der Wurzel den Besitz des Tokens verifiziert, während alle anderen Teilchecker verifizieren, dass ihre Komponente kein Token besitzt. Wir unterscheiden zwei Phasen: die Phase der Berechnung und die Phase der Prüfung, die alternierend ausgeführt werden.

17.2.1.4 *Berechnung*

Wenn die Komponente, die das Token besitzt (und Wurzel des Spannbaums ist), das Token an eines ihrer Kinder weiterreicht, dann wird der Zeuge verteilt aktualisiert. Dafür wird der Spannbaum dahingehend angepasst, dass nun das Kind, welches das Token erhalten hat, die neue Wurzel wird. Dabei bleiben genau dieselben Kanäle Teil des Spannbaums, die vorher schon zum Spannbaum gehörten. Die alte Wurzel wird ein Kind der neuen Wurzel. Die Abbildung 17.2 visualisiert die Aktualisierung.

Nur die Kanäle des Netzwerks, die zum Spannbaum gehören, sind dabei dargestellt. In der Abbildung ist u die alte Wurzel des Spannbaums und v die neue. Entsprechend aktualisieren die beiden Komponenten ihre Elternrelation so, dass u jetzt ein Kind von v wird. Für alle anderen Komponenten bleibt die Elternrelation unverändert. Alle Komponenten, die zum Teilbaum von v gehören, verringern ihre Distanz zur Wurzel um 1. Diese Komponenten sind in der Abbildung

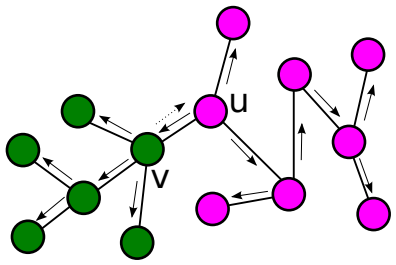


Abbildung 17.2: Darstellung des Zeugen: ein Spannbaum in einem Netzwerk – ohne die Darstellung weiterer Kanäle, die nicht zum Spannbaum gehören. Die Komponente u ist die Wurzel. Die Elternrelation ist durch Pfeile an den Kanälen dargestellt. Die Komponente v wird die neue Wurzel. Die nötige Aktualisierung der Elternrelation ist als gestrichelter Pfeil dargestellt. Die Komponenten, die zum Teilbaum von v gehören, sind grün gefärbt.

grün dargestellt. Alle anderen Komponenten erhöhen ihre Distanz um 1 zur Wurzel.

17.2.1.5 *Zeugen- & Verteilungseigenschaft*

Für die Zeugeneigenschaft ist folgendes zu zeigen. Wenn es einen Spannbaum gibt und nur die Wurzel ein Token besitzt, dann besitzt höchstens eine Komponente im Netzwerk ein Token. Die Verteilungseigenschaft ergibt sich direkt aus der Zertifizierung eines Spannbaums, wie wir sie in Abschnitt 9.3 vorgestellt haben.

17.2.1.6 *Übergang zwischen den Phasen*

Die Kommunikation funktioniert über Hin- und Rückwellen von Nachrichten in dem Spannbaum, die durch die neue Wurzel initiiert werden. Wir haben hierfür einen zertifizierenden Echo-Algorithmus in Abschnitt 9.6 beschrieben. Die Laufzeit der Phase der Berechnung ist abhängig von der Tiefe der Teilbäume und damit auch der Größe des Netzwerks. Die neue Wurzel wird durch den Echo-Algorithmus darüber informiert, wenn die Phase der Berechnung vorbei ist und leitet die Phase der Prüfung ein.

17.2.1.7 *Prüfung*

Der Checker ist hier genauso aufgebaut, wie wir ihn für zertifizierende terminierende verteilte Algorithmen beschrieben haben. Der entscheidende Unterschied ist, dass nicht einmalig nach der Terminierung geprüft wird, sondern jedes Mal, wenn das Token wandert.

Bei einem zertifizierenden verteilten Algorithmus, der terminiert, endet die Prüfung mit der Bekanntgabe der Entscheidung des verteilbaren Zeugenprädikats. Eine positive Entscheidung bildet den Übergang zu einer erneuten Phase der Berechnung. Eine negative Entscheidung bedeutet, das die Laufzeitverifikation fehlgeschlagen ist.

17.2.2 Allgemeine Ansätze zur Nicht-Terminierung

Während es bei der Zertifizierung verteilter Algorithmen, die terminieren, um die Korrektheit eines Eingabe-Ausgabe-Paares geht, ist bei der Nicht-Terminierung die Korrektheit eines Verhaltens während der Ausführung entscheidend. Diesen Unterschied finden wir auch zwischen sequentiellen Algorithmen und Datenstrukturen [AR05]. Eine Datenstruktur arbeitet reaktiv und wird deswegen auch den reaktiven Programmen zugeordnet [McC+11].

In [McC+11] liegt der Fokus auf zertifizierenden sequentiellen Algorithmen, aber es werden auch einige zertifizierende Datenstrukturen beschrieben [FM99; SM91; BS95]. Darüber hinaus sind uns wenige Arbeiten bekannt zu Zertifizierung und Nicht-Terminierung. Für die Zertifizierung verteilter Algorithmen, die nicht terminieren, sind zertifizierende Datenstrukturen folglich von Interesse.

17.2.2.1 *Zertifizierende Datenstrukturen*

Eine *Datenstruktur* bezeichnet eine bestimmte Art Daten zu speichern und verwalten. Datenstrukturen unterschieden sich untereinander dadurch, wie sie Daten strukturieren und welche Operationen sie auf den Daten erlauben.

Wir unterscheiden außerdem zwischen Datenstrukturen und abstrakten Datentypen. Ein *abstrakter Datentyp* ist ein mathematisches Objekt (zum Beispiel eine algebraische Struktur), welches das Verhalten von Operationen auf Daten spezifiziert. Ein abstrakter Datentyp spezifiziert Datenstrukturen. Eine Datenstruktur setzt einen abstrakten Datentyp um.

Im Gegensatz zu einem Algorithmus berechnet eine Datenstruktur keine Ausgabe. Sondern sie erhält eine *Eingabesequenz*, durch ein Programm, welches die Datenstruktur benutzt und berechnet eine *Ausgabesequenz*. Die Zertifizierung einer Datenstruktur bedeutet die Korrektheit der Ausgabesequenz für eine Eingabesequenz in Bezug auf einen abstrakten Datentyp mithilfe einer berechneten *Zeugensequenz* zur Laufzeit zu verifizieren.

17.2.2.2 *Online-Checker und Offline-Checker*

Für zertifizierende Datenstrukturen ist es sinnvoll zwischen *Offline*-Checkern und *Online*-Checkern zu unterschieden [FM99]. Ein Offline-Checker verifiziert die Korrektheit einer Ausgabe (aus einer Ausgabesequenz) unmittelbar. Wir haben bisher Offline-Checker betrachtet. Das ist sinnvoll, da bei einem terminierenden Algorithmus nach der Terminierung alle Informationen zur Verfügung stehen. Wenn es nur eine Ausgabe, also keine Ausgabesequenz gibt, ist nur die Betrachtung von Offline-Checkern sinnvoll.

Wenn wir jedoch eine Ausgabesequenz einer Datenstruktur betrachten, dann sind zu jedem Zeitpunkt einer Ausgabe der Sequenz potenziell noch nicht alle Informationen vorhanden. Mit jeder nachfolgenden Ausgabe können noch weitere Informationen über das Verhalten der Datenstruktur hinzukommen. Online-Checker verifizieren eine Ausgabe einer Sequenz verzögert. Ein Online-Checker wartet nach einer Ausgabe a weitere Ausgaben der Sequenz ab, um zu entscheiden, ob die Ausgabesequenz bis a korrekt war. Dabei unterscheiden sich Online-Checker darin, wie viele Ausgaben sie nach a noch abwarten müssen, um eine Aussage zur Korrektheit der Sequenz bis a machen zu können.

Für einige Datenstrukturen gibt es keinen Offline-Checker, die nicht genau die gleichen Berechnungen machen müsste wie die Datenstruktur selbst; ein Beispiel dafür ist eine Prioritätswarteschlange [FM99]. Sie geben jedoch einen Online-Checker mit einer oberen Schranke für die Verzögerung an, die in Abhängigkeit der Anzahl der gespeicherten Elemente wächst.

17.2.2.3 *Inspiration für Nicht-Terminierung*

Wir haben in der Fallstudie zum Mutex-Algorithmus (siehe vorigen Abschnitt 17.2.1) einen Offline-Checker gesehen. Unser Checker ist dabei vor allem durch die Evaluation mit ihrer notwendigen Hin- und Rückwelle von Nachrichten aufwändig (siehe Kapitel 10 für die Evaluation).

Ein Online-Checker hingegen könnte für eine bestimmte Anzahl k von Phasen der Prüfung auf die Evaluation verzichten. Jeder Teilchecker entscheidet dann die lokalen Prädikate für seine Komponente. Die Evaluation findet dann für alle ausgewerteten lokalen Prädikate der letzten k Phasen gemeinsam in Phase $k+1$ statt. Eine interessante Frage ist dabei, nach welchem Kriterium k zu entscheiden ist.

17.2.2.4 *Prüfung und Asynchronität*

Synchrone Netzwerke arbeiten runden-basiert. Es können jeweils mehrere Runden für die Phasen der Berechnung und der Prüfung nötig sein.

In asynchronen Netzwerken ist es im Allgemeinen eine Herausforderung den Zeitpunkt der Prüfung zu bestimmen. Die Teilchecker könnten hierfür verteilte Algorithmen zur Synchronisierung für die Prüfung benutzen oder auch sogenannte Snapshot-Algorithmen, die einen globalen Zustand eines Netzwerks festhalten.

Teil VII

DISKUSSION

Wir diskutieren in diesem Teil die Ergebnisse der vorliegenden Arbeit. In Kapitel 18 fassen wir diese Ergebnisse zunächst zusammen.

In Kapitel 19 geben wir einen Ausblick auf sich anschließende zukünftige Forschungsfragen.

18 ZUSAMMENFASSUNG

In diesem Kapitel fassen wir die Ergebnisse der vorliegenden Arbeit zusammen. Dabei greifen wir die von uns herausgestellten Herausforderungen bei der Übertragung des Konzepts zertifizierender Algorithmen auf verteilte Algorithmen auf (siehe Abschnitt 1.2.2).

In Abschnitt 18.1 geben wir einen Überblick über die Ergebnisse der von uns entwickelten Methode, das Konzept zertifizierende sequentielle Algorithmen auf terminierende verteilte Algorithmen zu übertragen. Wir fassen unsere Ergebnisse zur formalen Instanzverifikation in Abschnitt 18.2 zusammen.

In Abschnitt 18.3 besprechen wir unsere Ergebnisse zur Betrachtung weiterer Methoden der Übertragung des Konzepts: lokale Korrektheit und Nicht-Terminierung.

18.1 ZERTIFIZIERENDE TERMINIERENDE VERTEILTE ALGORITHMEN

Methode der Übertragung. Wir haben eine Methode vorgestellt, das Konzept zertifizierender sequentieller Algorithmen auf terminierende verteilte Algorithmen zu übertragen. Die Übertragung resultiert in einer Klasse zertifizierender verteilter Algorithmen,

- die ihr verteiltes Eingabe-Ausgabe-Paar verifizieren,
- einen verteilten Zeugen mit jeder Ausgabe berechnen und
- ein verteilbares Zeugenprädikat besitzen,
- das wiederum durch einen verteilten Checker entschieden wird.

Verteilte Zeugen & Konsistenz. Eine allgemeine Herausforderung verteilter Algorithmen ist, dass jede Komponente unvollständiges Wissen über das Netzwerk, sowie eine verteilte Eingabe und Ausgabe hat. Diese Herausforderung führt für verteilte Zeugen zum Problem der Konsistenz. Jede Komponente berechnet einen Teilzeugen. Dabei weiß sie nicht, ob dieser mit den Teilzeugen der anderen Komponenten konsistent ist. Wir haben das Problem der Konsistenz untersucht und mit zusammenhängenden Zeugen eine allgemeine Konstruktion für Zeugen aufgezeigt, deren Konsistenz lokal in den Nachbarschaften eines Netzwerks prüfbar ist.

Verteilbare Zeugenprädikate und Teilchecker. Des Weiteren haben wir verteilbare Zeugenprädikate eingeführt und untersucht. Ein verteilbares Prädikat kann entschieden werden, indem jeder Teilchecker unabhängig lokale Prädikate für seine Komponenten entscheidet. Durch die Einführung verteilbarer Zeugenprädikate haben wir eine verteilte Laufzeitverifikation erreicht.

Checker und Kommunikation. Wir haben uns mit der Herausforderung der Kommunikation in Netzwerken im Kontext von Checkern beschäftigt. Wir haben gezeigt, dass ein sequentieller Checker nicht zu den Gegebenheiten eines Netzwerks passt und deshalb eine verteilte Architektur für Checker entworfen. Die Kommunikation für die Aufgabe der Evaluation eines verteilbaren Zeugenprädikats läuft immer nach dem gleichen Schema ab. Für die Aufgabe der Konsistenzprüfung haben wir zum einen eine Konsistenzprüfung für beliebige Zeugen entwickelt, für die im gesamten Netzwerk Kommunikation anfällt und zum anderen eine lokale Konsistenzprüfung zusammenhängender Zeugen für die nur in Nachbarschaften kommuniziert wird.

Virtuelle Netzwerke und Invasivität. Anhand der industriellen Fallstudie der zertifizierenden verteilten Auktion für Transportroboter in einer Fabrik haben wir uns zum einen mit Multi-Agenten-Systemen beschäftigt und zum anderen den Einsatz virtueller Netzwerke zur Reduktion der Invasivität des Checkers untersucht. Wir konnten anhand der Fallstudie nicht nur eine Reduktion aufzeigen, sondern die Ergebnisse sind auch auf jedes Netzwerk übertragbar, dessen Topologie bekannt ist.

Entwicklung zertifizierender verteilter Algorithmen. Wir haben zertifizierende verteilte Algorithmen aus einem softwaretechnischen Blickwinkel heraus betrachtet. Zur Analyse der Kerngedanken Verteiltheit, Gleichheit und Lokalität bei der Zertifizierung haben wir zentralisierte Zeugen, gleichverteilte Zeugen und beschränkte Zeugen untersucht. Wir haben außerdem Kriterien zur Güte einer Zertifizierung, sowie Entwurfsmuster für zertifizierende verteilte Algorithmen entwickelt.

Auswertung der Ziele bei der Übertragung. Wir haben für eine Übertragung des Konzepts zertifizierender Algorithmen auf verteilte Algorithmen zwei Ziele motiviert (siehe Kapitel 1). Ein Ziel war, das Konzept so zu übertragen, dass es nah am ursprünglichen Konzept sequentieller Algorithmen ist. Wir haben uns bei dieser Übertragung auf terminierende verteilte Algorithmen beschränkt, wodurch uns eine direktere Übertragung gelang. Zertifizierende verteilte Algorithmen sind ein Spezialfall zertifizierender sequentieller Algorithmen, mit vergleichbaren Rollen für Zeuge, Zeugenprädikat und Checker.

Ein weiteres Ziel war, das Konzept so zu übertragen, dass es zu den Gegebenheiten eines verteilten Systems passt. Wir denken, das uns dies gelungen ist, denn Zeugen werden verteilt berechnet und Zeugen-

prädikate verteilt verifiziert. Wir denken außerdem, dass die verschiedenen Fallstudien zeigen, dass die entwickelte Klasse zertifizierender verteilter Algorithmen diesem Anspruch gerecht wird.

Verteilte Laufzeitverifikation. Unsere Methode der Übertragung resultiert in einer verteilten Laufzeitverifikation – einem Gebiet, zu dem es bisher nur wenige Arbeiten gibt. Die Konferenz „International Conference on Runtime Verification“ fand im Jahr 2010 erstmalig statt und während der Fokus seit Beginn auf der Laufzeitverifikation verteilter Systeme liegt, beschreiben die meisten Arbeiten dort sequentielle Ansätze mit einem zentralen Monitor. So beschäftigt sich im Jahr 2019 bei der Konferenz nur eine Arbeit mit verteilter Laufzeitverifikation [FM19]. In den Jahren 2016 und 2017, in denen die Autorin der vorliegenden Arbeit teilnahm, war die Situation vergleichbar und wurde in den abschließenden Diskussionen thematisiert [FS16; LR17].

18.2 FORMALE INSTANZVERIFIKATION

Methodik und Framework. Wir haben eine Methodik zur formalen Instanzverifikation für zertifizierende verteilte Algorithmen entwickelt und diese in einem Framework für den Beweisassistenten COQ umgesetzt. Verifizierte Checker können dabei auf einem realen Netzwerk laufen und sogar verifiziert fehlertolerant gestaltet werden.

Fallstudien. Wir haben das Framework an drei Fallstudien illustriert. Des Weiteren haben wir die lokale und die allgemeine Konsistenzprüfung für das Framework in COQ implementiert.

18.3 LOKALE KORREKTHEIT & NICHT-TERMINIERUNG

Wir haben uns im Kontext der lokalen Korrektheit und der Nicht-Terminierung auch mit weiteren Methode der Übertragung des Konzepts zertifizierender Algorithmen auf verteilte Algorithmen befasst.

Lokale Korrektheit. Wir haben das Problem der lokalen Korrektheit formuliert und durch das Szenario, einer fehlschlagenden Laufzeitverifikation, motiviert. Anhand einer Fallstudie haben wir aufgezeigt, wie die Zertifizierung der gesamten Ausgabe des Netzwerks genutzt werden kann, um die lokale Korrektheit für Teilausgaben einiger Komponenten zu entscheiden.

Nicht-Terminierung. Wir haben anhand einer Fallstudie eines Mutex-Algorithmus analysiert, wie Zertifizierung für verteilte Algorithmen, die nicht terminieren, gelingen kann.

19 AUSBLICK

In diesem Kapitel diskutieren wir zukünftige Forschungsfragen. Wir haben für jeden Themenkomplex in dieser Arbeit bereits „kleinere“ weiterführende Forschungsfragen diskutiert, die sich direkt an das jeweilige Thema anschlossen. In diesem Ausblick betrachten wir nur Forschungsfragen, die neue Themenbereiche eröffnen.

In Abschnitt 19.1 betrachten wir, wie eine Methode der Übertragung des Konzepts auf nicht terminierende verteilte Algorithmen aussehen könnte. In Abschnitt 19.2 diskutieren wir allgemein eine Übertragung auf andere Klassen verteilte Systeme als die, von uns untersuchten, Netzwerke.

19.1 ZERTIFIZIERUNG & NICHT-TERMINIERUNG

Wir haben in Abschnitt 17.2 einen zertifizierenden Mutex-Algorithmus betrachtet. Die dabei gewonnen ersten Ergebnisse werfen die Forschungsfrage auf, wie eine Methode zur Übertragung zertifizierender Algorithmen auf verteilte Algorithmen aussehen könnte, die nicht terminieren. Die Autorin der vorliegenden Arbeit plant sich zukünftig mit dieser Forschungsfrage zu beschäftigen.

Methode einer Übertragung. Genau wie für terminierende verteilte Algorithmen, interessieren wir uns für eine verteilte Laufzeitverifikation für verteilte Algorithmen, die nicht terminieren. Daraus folgt, dass es sich anbietet, verteilbare Prädikate auch für deren Zertifizierung zu untersuchen.

Ein Fallstudie dafür haben wir mit dem zertifizierenden Mutex-Algorithmus bereits gesehen. Wir erwarten, dass sich einige der Ideen dabei verallgemeinern lassen. Der betrachtete Mutex-Algorithmus zertifiziert mithilfe eines verteilten Zeugen und einem verteilbaren Zeugenprädikat eine Sicherheitseigenschaft: genau eine Komponente in einem Netzwerk hat Zugriff auf eine Ressource. Dabei unterscheidet sich die Zertifizierung jedoch von der vorgestellten Klasse zertifizierender terminierender verteilter Algorithmen.

Zeugensequenz. Der zertifizierende Mutex-Algorithmus berechnet eine Zeugensequenz: einen Zeugen für jeden Zustand, bei dem der Zugriff von einer Komponente zu einer anderen Komponente wechselt.

Die Zeugen der Sequenz bauen dabei aufeinander auf. Wir erwarten, dass wir häufig eine Zeugensequenz finden können, sodass ein Zeuge der Sequenz dem vorigen Zeugen ähnelt. Das wäre insbesondere interessant für eine inkrementelle verteilte Berechnung der Zeugensequenz. Darüber hinaus könnten wir die Ähnlichkeit zweier aufeinander folgenden Zeugen auch nutzen, um die Prüfung der Konsistenz eines Zeugen zu optimieren. Es müssen dann immer nur die hinzugekommenen Informationen eines Zeugen auf Konsistenz geprüft werden.

In [DN17] wird eine inkrementelle Berechnung einer Zeugensequenz in einem verteilten System beschrieben, allerdings für einen sequentiellen Checker. Da der Checker zudem zentral ist, tritt das Problem der Konsistenz hierbei nicht auf.

Zeugenprädikat für Zustandssequenz. Der zertifizierende Mutex-Algorithmus unterscheidet sich auch deshalb von den vorgestellten zertifizierenden verteilten Algorithmen, weil sein Zeugenprädikat für eine Sequenz von Zuständen gelten muss; analog zur Zeugensequenz immer dann, wenn der Zugriff von einer Komponente zur einer anderen wechselt. In dem Beispiel ist es immer dasselbe verteilbare Prädikat, das im Netzwerk gelten muss.

Wir können uns jedoch vorstellen, dass ein Zeugenprädikat allgemeiner auch durch eine Kombination verteilbarer Prädikate impliziert wird. Bei der Zertifizierung terminierender verteilter Algorithmen haben wir bereits verteilbare Prädikate kombiniert, die gemeinsam das Zeugenprädikat implizieren. Bei der Nicht-Terminierung können wir uns darüber hinaus andere interessante Kombinationsmöglichkeiten vorstellen, nämlich über die Sequenz der Zustände. Zum Beispiel könnten zwei verteilbare Prädikate, die alternierend für eine Zustandssequenz gelten, ein Zeugenprädikat implizieren. Andere Schemata als alternierend sind hierbei denkbar.

Online-Checker. Für terminierende verteilte Algorithmen genügte es Offline-Checker zu betrachten und auch für den zertifizierenden Mutex-Algorithmus haben wir einen Offline-Checker angegeben. Wir haben jedoch skizziert, wie ein effizienterer Online-Checker aussehen könnte. Darüber hinaus gibt es bereits Betrachtungen zu Online-Checkern für zertifizierende Datenstrukturen. Für einige Datenstrukturen gibt es keinen Offline-Checker, der nicht genau die gleichen Berechnungen durchführen müsste wie die Datenstruktur selbst; ein Beispiel dafür ist eine Prioritätswarteschlange [FM99]. Wir denken deswegen, dass eine Untersuchung von Online-Checkern für die Zertifizierung verteilter Algorithmen, die nicht terminieren, vielversprechend ist.

Entwurfsmuster. Der zertifizierende Mutex-Algorithmus hat die Frage aufgeworfen, ob sich selbst-stabilisierende Algorithmen, die einen Mechanismus zur lokalen Reparatur besitzen, als besonders guter Ausgangspunkt für die Entwicklung einer zertifizierenden Variante anbieten [Var93; Awe+94]. Es scheint als ließe sich aus der lokalen

Reparatur besonders gut eine inkrementelle Berechnung einer Zeugensequenz ableiten.

Zur Entwicklung verteilter Online-Checker finden wir vermutlich Inspiration aus den Arbeiten zu Online-Algorithmen, die es für sequentielle Algorithmen gibt [AL99; AP19].

Formale Instanzverifikation. Das von uns entwickelte Framework zur formalen Instanzverifikation eignet sich, um es für die Zertifizierung verteilter Algorithmen, die nicht terminieren, auszubauen. Für das Netzwerkmodell müsste dafür nichts angepasst werden. Für das Beweisen der Zeugeneigenschaft bräuchten wir zusätzlich eine Formalisierung der Sequenzen von Zuständen. Für die Verifikation der Checker müssten wir vermutlich die Formalisierung der Zustandssequenz mit den Zustandsübergängen der Netzwerksemantik in Verdi verknüpfen.

19.2 WEITERE VERTEILTE SYSTEME

Wir haben uns in der vorliegenden Arbeit mit Netzwerken beschäftigt, die statisch sind. Entsprechend wäre es interessant dynamische Netzwerke, sowie verteilte Systeme fernab von Netzwerken für die Zertifizierung zu untersuchen.

Dynamische Netzwerke. Eine interessante Richtung ist es, dynamische Netzwerke bezüglich einer Zertifizierung zu untersuchen. Falls eine maximale Anzahl an Komponenten oder Kanälen bekannt ist, die wegfallen können, dann könnten Zeugen eine gewisse Redundanz aufweisen. Für welche Probleme das möglich ist, ist dabei vollkommen offen.

Eine Inspiration könnte auch die Super-Stabilisierung aus dem Bereich der Selbst-Stabilisierung sein: ein selbst-stabilisierender Algorithmus wird dabei mit einem verteilten Algorithmus kombiniert, der für ein dynamisches Netzwerk ausgelegt ist [DH97]. Entsprechend sind verteilte Algorithmen für dynamische Netzwerke von Interesse [KLO10].

Weitere verteilte Systeme. Wir haben uns in der vorliegenden Arbeit hauptsächlich mit Netzwerken und nebensächlich mit Multi-Agenten-Systemen beschäftigt. Auch eine Beschäftigung mit verteilten Systemen mit geteiltem Speicher ist interessant. Inwiefern unsere Ergebnisse übertragbar sind, ist dabei unklar. Wir erwarten jedoch, dass das Problem der Konsistenz entfällt. Der Grund dafür ist, die Herausforderung des unvollständigen Wissens einer Komponenten über einen Zeugen entfällt.

Ganz andere Richtungen bietet die Untersuchung von zum Beispiel Population-Protocols [AR09b], Grid-Computing [Ber+03], Map-Redu-

ce [DG08] und vielen weiteren verteilten Systemen, die Infrastrukturen für verteiltes Rechnen bieten.

Teil VIII

ANHANG & LITERATUR

MATHEMATISCHE NOTATIONEN

Wir führen in diesem Kapitel in Kürze die mathematischen Notationen ein, die wir ansonsten ohne Einführung in der vorliegenden Arbeit verwenden. Wir geben dabei keine Erklärungen oder Beispiele an. Das Kapitel zeigt lediglich auf, in welchem Kontext wir welche Symbole verwenden.

MENGEN

Wir nehmen an, dass der Begriff einer Menge definiert ist und verwenden die üblichen Notationen: $\in, \ni, \notin, \emptyset, \subset, \subseteq, \supset, \supseteq, \cup, \cap, \setminus$.

Wir beschreiben Mengen auch durch $\{e|P(e)\}$, wobei e ein Element ist und P ein Prädikat.

Besondere Mengen, die wir voraussetzen, sind die natürlichen Zahlen $\mathbb{N}$, die positiven natürlichen Zahlen $\mathbb{N}_+$, die ganzen Zahlen $\mathbb{Z}$, die reellen Zahlen $\mathbb{R}$, sowie die positiven reellen Zahlen $\mathbb{R}_+$.

Die Kardinalität einer Menge M ist $|M|$ und ihre Potenzmenge $P(M)$.

ARITHMETIK, FUNKTIONEN UND RELATIONEN

Wir notieren eine Funktione über f über Mengen A, B wie folgt: $f : A \to B$.

Wir setzen die arithmetischen Funktionen $+, -, *$ wie üblich voraus.

Wir schreiben Tupel der Größe $n \in \mathbb{N}$ als $(x_1, x_2, ..., x_n)$.

Wir setzen die Relationen $=, \leqslant, <, \geqslant, >$ wie üblich voraus.

LOGIK

Wir setzen Aussagenlogik und Prädikatenlogik voraus. Wir benutzen die üblichen Notationen: $\neg, \vee, \wedge, \exists, \forall, \to$.

Im Sinne eines Kalküls des natürlichen Schließens benutzen wir die üblichen Notationen. Aus Aussage A ist Aussage B ableitbar: $\frac{A}{B}$. Im Kontext des Systems Γ ist die Aussage A beweisbar: $\Gamma \vdash A$.

LANDAU-NOTATION

Wir setzen die Landau-Notation wie üblich voraus: $\mathcal{O}, \Omega, \Theta, o, \omega$.

LITERATUR

[Aba+13] Martin Abadi u. a. „Formal Analysis of a Distributed Algorithm for Tracking Progress“. In: *FMOODS-FORTE'13: 15th Formal Methods for Open Object-Based Distributed Systems and 33nd Formal Techniques for Networked and Distributed Systems*. Springer, 2013.

[ABG18] C. Aiswarya, Benedikt Bollig und Paul Gastin. „An automata-theoretic approach to the verification of distributed algorithms“. In: *Information and Computation* 259 (2018). 22nd International Symposium on Temporal Representation and Reasoning, S. 305 –327.

[ACD16] Karine Altisen, Pierre Corbineau und Stéphane Devismes. „A Framework for Certified Self-Stabilization“. In: *Formal Techniques for Distributed Objects, Components, and Systems: 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*. Hrsg. von Elvira Albert und Ivan Lanese. Cham: Springer International Publishing, 2016, S. 36–51.

[AFP13] Heger Arfaoui, Pierre Fraigniaud und Andrzej Pelc. „Local Decision and Verification with Bounded-Size Outputs“. In: *Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*. Hrsg. von Teruo Higashino u. a. Cham: Springer International Publishing, 2013, S. 133–147.

[Agd] *Homepage des Beweisassistenten Agda*. http://wiki.portal.chalmers.se/agda/pmwiki.php.

[Aki15] Samira Akili. *Simulation Zertifizierender Netzwerkalgorithmen*. Humboldt-Universität zu Berlin. 2015.

[Aki18] Samira Akili. *Verteilte Prüfung der Konsistenz im Rahmen eines Verifikations-Frameworks für Zertifizierende Verteilte Algorithmen*. Humboldt-Universität zu Berlin. 2018.

[AL19] Samira Akili und Felix Lorenz. „Towards runtime verification of collaborative embedded systems“. In: *SICS Software-Intensive Cyber-Physical Systems* 34.4 (2019), S. 225–236.

[AL94] Nancy M. Amato und Michael C. Loui. „Checking Linked Data Structures“. In: *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing (FTCS)*. 1994, S. 164–173.

[AL99] Susanne Albers und Stefano Leonardi. „On-line Algorithms“. In: *ACM Comput. Surv.* 31.3es (1999), S. 4.

[Alb+12] Elvira Albert u. a. „Certificate Size Reduction in Abstraction-carrying Code*“. In: *Theory Pract. Log. Program.* 12.3 (Mai 2012), S. 283–318.

[Alk+11] Eyad Alkassar u. a. „Verification of Certifying Computations“. In: *Computer Aided Verification*. Hrsg. von Ganesh Gopalakrishnan und Shaz Qadeer. Bd. 6806. Lecture Notes in Computer Science. Springer, 2011, S. 67–82.

[Alk+14] Eyad Alkassar u. a. „A Framework for the Verification of Certifying Computations“. In: *Journal of Automated Reasoning* 52.3 (2014), S. 241–273.

[Ami+18] Benjamin Aminof u. a. „Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction“. In: *Verification, Model Checking, and Abstract Interpretation*. Hrsg. von Isil Dillig und Jens Palsberg. Cham: Springer International Publishing, 2018, S. 1–24.

[AP19] Susanne Albers und Achim Passen. „New Online Algorithms for Story Scheduling in Web Advertising“. In: *Algorithmica* 81.1 (2019), S. 1–25.

[AR05] Konstantine Arkoudas und Martin C. Rinard. „Deductive Runtime Certification.“ In: *Electronic Notes in Theoretical Computer Science* 113 (2005), S. 45–63.

[AR09a] James Aspnes und Eric Ruppert. „An Introduction to Population Protocols“. In: *Middleware for Network Eccentric and Mobile Applications*. Hrsg. von Benoît Garbinato, Hugo Miranda und Luís Rodrigues. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 97–120.

[AR09b] James Aspnes und Eric Ruppert. „An Introduction to Population Protocols“. In: *Middleware for Network Eccentric and Mobile Applications*. Hrsg. von Benoît Garbinato, Hugo Miranda und Luís Rodrigues. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 97–120.

[AS98] Sanjeev Arora und Shmuel Safra. „Probabilistic Checking of Proofs: A New Characterization of NP“. In: *J. ACM* 45.1 (Jan. 1998), S. 70–122.

[Ash16] David Asher. *Verifikation eines zertifizierenden verteilten Algorithmus*. Humboldt-Universität zu Berlin. 2016.

[AV19] Samira Akili und Kim Völlinger. „Case Study on Certifying Distributed Algorithms: Reducing Intrusiveness“. In: *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers*. Hrsg. von Hossein Hojjat und Mieke Massink. Bd. 11761. Lecture Notes in Computer Science. Springer, 2019, S. 179–185.

[AW04] Hagit Attiya und Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. USA: John Wiley & Sons, Inc., 2004.

[Awe+94] Baruch Awerbuch u. a. „Self-Stabilization by Local Checking and Global Reset (Extended Abstract)“. In: *Proceedings of the 8th International Workshop on Distributed Algorithms*. WDAG 94. London, UK, UK: Springer-Verlag, 1994, S. 326–339.

[Azi11] Haris Aziz. „Review of graph theory: a problem oriented approach by Daniel Marcus“. In: *SIGACT News* 42 (Dez. 2011), S. 31–32.

[Bae19] Sammie Bae. „Dynamic Programming“. In: *JavaScript Data Structures and Algorithms: An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals*. Berkeley, CA: Apress, 2019, S. 321–338.

[Bar+15] E. T. Barr u. a. „The Oracle Problem in Software Testing: A Survey“. In: *IEEE Transactions on Software Engineering* 41.5 (2015), S. 507–525.

[Ber+03] Fran Berman u. a. *Grid Computing: Making the Global Infrastructure a Reality*. USA: John Wiley und Sons, Inc., 2003.

[Ber+19] Idan Berkovits u. a. „Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics“. In: *Computer Aided Verification*. Hrsg. von Isil Dillig und Serdar Tasiran. Cham: Springer International Publishing, 2019, S. 245–266.

[Ber82] Dimitri Bertsekas. „Distributed dynamic programming“. In: Bd. 37. Juni 1982, S. 774 –779.

[Bey+15] Dirk Beyer u. a. „Witness Validation and Stepwise Testification across Software Verifiers“. In: *Proceedings of the 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE) 2015, Bergamo, Italy, August 31 - September 4*. Hrsg. von E. Di Nitto, M. Harman und P. Heymans. ACM, New York, 2015, S. 721–733.

[Bey+16] Dirk Beyer u. a. „Correctness Witnesses: Exchanging Verification Results Between Verifiers“. In: *Proc. FSE*. ACM, 2016, S. 326–337.

[Bey17] Dirk Beyer. „Software Verification with Validation of Results (Report on SV-COMP 2017)“. In: *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2017, Uppsala, Sweden, April 22-29)*. Hrsg. von Axel Legay und Tiziana Margaria. LNCS 10206. Springer-Verlag, Heidelberg, 2017, S. 331–349.

[BF17] Alkida Balliu und Pierre Fraigniaud. „Certification of Compact Low-Stretch Routing Schemes“. In: *31st International Symposium on Distributed Computing (DISC 2017)*. Hrsg. von Andréa W. Richa. Bd. 91. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 6:1–6:16.

[BG11] Jan Olaf Blech und Benjamin Grégoire. „Certifying compilers using higher-order theorem provers as certificate checkers“. In: *Formal Methods in System Design* 38.1 (2011), S. 33–61.

[BHS09] Daniel Bruce, Chinh T. Hoang und Joe Sawada. „A certifying algorithm for 3-colorability of P5-free graphs“. In: *The 20th International Symposium on Algorithms and Computation (ISAAC 2009), LNCS 5878*. 2009.

[BJP06] Frédéric Besson, Thomas Jensen und David Pichardie. „Proof-carrying Code from Certified Abstract Interpretation and Fixpoint Compression“. In: *Theor. Comput. Sci.* 364.3 (Nov. 2006), S. 273–291.

[BK12] Frédéric Blanqui und Adam Koprowski. „CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates“. In: *CoRR* abs/1202.6473 (2012). arXiv: 1202.6473.

[BK18] Leonid Barenboim und Victor Khazanov. „Distributed Symmetry-Breaking Algorithms for Congested Cliques“. In: *CoRR* abs/1802.07209 (2018).

[BK95] Manuel Blum und Sampath Kannan. „Designing Programs that Check Their Work“. In: *Jornal of the ACM* 42.1 (1995), S. 269–291.

[BKZ15] David Basin, Felix Klaedtke und Eugen Zalinescu. „Failure-aware Runtime Verification of Distributed Systems“. In: *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*. Hrsg. von Prahladh Harsha und G. Ramalingam. Bd. 45. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, S. 590–603.

[Blu+94] M. Blum u. a. „Checking the correctness of memories“. In: *Algorithmica* 12.2 (1994), S. 225–244.

[Blu93] Manuel Blum. „Program result checking: A new approach to making programs more reliable“. In: *Automata, Languages and Programming: 20th International Colloquium, ICALP 93 Lund, Sweden, July 5–9, 1993 Proceedings*. Hrsg. von Andrzej Lingas, Rolf Karlsson und Svante Carlsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, S. 1–14.

[BLW08] Sascha Böhme, K. Rustan M. Leino und Burkhart Wolff. „HOL-Boogie – An Interactive Prover for the Boogie Program-Verifier“. In: *Theorem Proving in Higher Order Logics*. Hrsg. von Otmane Ait Mohamed, César Muñoz und Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 150–166.

[Bol19] Alexander Boll. *Formale Instanzverifikation zertifizierener Algorithmen: Fallstudie Zweifärbbarkeit & alternative Konsistenzprüfung*. Humboldt-Universität zu Berlin. 2019.

[BP09] Jan Olaf Blech und Michaël Périn. „Certifying Deadlock-freedom for BIP Models“. In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '09. Nice, France: ACM, 2009, S. 61–70.

[BS01] Mike Barnett und Wolfram Schulte. „Spying on Components: A Runtime Verification Technique“. In: *Proceedings of the Workshop on Specification and Verification of Component-Based Systems OOPSLA 2001*. 2001.

[BS95] Jonathan D. Bright und Gregory F. Sullivan. „On-line Error Monitoring for Several Data Structures“. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, FTCS-25. Digest of Papers* (1995).

[BSM97] Jonathan D. Bright, Gregory F. Sullivan und Gerald M. Masson. „A Formally Verified Sorting Certifier“. In: *IEEE Trans. Comput.* 46.12 (1997), S. 1304–1312.

[BW97] Manuel Blum und Hal Wasserman. „Software Reliability via Run-Time Result-Checking“. In: *JOURNAL OF THE ACM* 44 (6 1997), S. 826–849.

[CDH13] Derek G. Corneil, Barnaby Dalton und Michel Habib. „LDFS-Based Certifying Algorithm for the Minimum Path Cover Problem on Cocomparability Graphs“. In: *SIAM Journal on Computing* 42.3 (2013), S. 792–807.

[CFM09] Pierre Castéran, Vincent Filou und Mohamed Mosbah. „Certifying Distributed Algorithms by Embedding Local Computation Systems in the Coq Proof Assistant“. In: *Symbolic Computation in Software Science (SCSS 2009)*. 2009.

[CH+16] Keren Censor-Hillel u. a. „Fast Distributed Algorithms for Testing Graph Properties“. In: *Distributed Computing*. Hrsg. von Cyril Gavoille und David Ilcinkas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, S. 43–56.

[Cha06] Amine Chaieb. „Proof-producing Program Analysis“. In: *Proceedings of the Third International Conference on Theoretical Aspects of Computing*. ICTAC'06. Tunis, Tunisia: Springer-Verlag, 2006, S. 287–301.

[Cha11] Arthur Charguéraud. „Characteristic Formulae for the Verification of Imperative Programs“. In: *SIGPLAN Not.* 46.9 (Sep. 2011), S. 418–430.

[Che+18] Ran Chen u. a. „Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle“. In: (Okt. 2018).

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[CM12] Meixian Chen und Jean-François Monin. „Formal Verification of Netlog Protocols“. In: *TASE*. Hrsg. von Tiziana Margaria, Zongyan Qiu und Hongli Yang. Beijing, China: IEEE, Juli 2012.

[Coqa] *Homepage des Beweisassistenten Coq*. `http://coq.inria.fr/`.

[Coqb] *Index der Coq-Taktiken*. `http://coq.inria.fr/refman/tactic-index.html`.

[Cor+01] Thomas H. Cormen u. a. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001.

[Cor13] Pierre-Emmanuel Cornilleau. „Certification of static analysis in many-sorted first-order logic“. Theses. École normale supérieure de Cachan - ENS Cachan, März 2013.

[CW00] Karl Crary und Stephnie Weirich. „Resource Bound Certification“. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA: ACM, 2000, S. 184–198.

[Del00] D. Delahaye. *A Tactic Language for the System Coq*. 2000.

[DG08] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *Commun. ACM* 51.1 (Jan. 2008), S. 107–113.

[DH97] Shlomi Dolev und Ted Herman. „Superstabilizing Protocols for Dynamic Distributed Systems“. In: *Chicago Journal of Theoretical Computer Science* 1997.4 (1997).

[DKJ18] Ali Dorri, Salil Kanhere und Raja Jurdak. „Multi-Agent Systems: A survey“. In: *IEEE Access* (Apr. 2018), S. 1–1.

[DN17] Chaoqiang Deng und Kedar S. Namjoshi. „Witnessing Network Transformations“. In: *Runtime Verification: 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. Hrsg. von Shuvendu Lahiri und Giles Reger. Cham: Springer International Publishing, 2017, S. 155–171.

[Dolo0] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[Drä+10] Klaus Dräger u. a. „SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems“. In: *Tools and Algorithms for the Construction and Analysis of Systems: 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Hrsg. von Javier Esparza und Rupak Majumdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 271–274.

[DS+11] Atish Das Sarma u. a. „Distributed Verification and Hardness of Distributed Approximation“. In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*. STOC '11. San Jose, California, USA: ACM, 2011, S. 363–372.

[DS80] E.W. Dijkstra und C.S. Scholten. „Termination detection for diffusing computations“. English. In: *Information Processing Letters* 11.1 (1980), S. 1–4.

[DSMS07] Patricio Domingues, Bruno Sousa und Luis Moura Silva. „Sabotage-tolerance and Trust Management in Desktop Grid Computing“. In: *Future Gener. Comput. Syst.* 23.7 (Aug. 2007), S. 904–912.

[Dup01] Jean Duprat. „A Coq toolkit for graph theory“. In: *Rapport de recherche, École normale supérieure de Lyon* 15 (2001).

[Dyc92] R. Dyckhoff. „Contraction-free sequent calculi for intuitionistic logic“. In: *The Journal of Symbolic Logic* 57 (1992).

[Erc13] Kayhan Erciyes. *Distributed Graph Algorithms for Computer Networks*. Juni 2013.

[Erg+98] Funda Ergün u. a. „Spot-checkers“. In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: ACM, 1998, S. 259–268.

[FB14] Alexis Fouilhe und Sylvain Boulmé. „A Certifying Frontend for (Sub)polyhedral Abstract Domains“. In: *Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. Hrsg. von Dimitra Giannakopoulou und Daniel Kroening. Cham: Springer International Publishing, 2014, S. 200–215.

[FF16] Laurent Feuilloley und Pierre Fraigniaud. „Survey of Distributed Decision“. In: *Bulletin of the EATCS* 119 (2016).

[FGY96] Yair Frankel, Peter Gemmell und Moti Yung. „Witness-based Cryptographic Program Checking and Robust Function Sharing“. In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, S. 499–508.

[FKP13] Pierre Fraigniaud, Amos Korman und David Peleg. „Towards a Complexity Theory for Local Distributed Computing“. In: *J. ACM* 60.5 (Okt. 2013), 35:1–35:26.

[FM19] Bernd Finkbeiner und Leonardo Mariani, Hrsg. *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Bd. 11757. Lecture Notes in Computer Science. Springer, 2019.

[FM99] Ulrich Finkler und Kurt Mehlhorn. „Checking Priority Queues“. In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '99. Baltimore, Maryland, USA: Society for Industrial und Applied Mathematics, 1999, S. 901–902.

[FMP13] Alexis Fouilhe, David Monniaux und Michaël Périn. „Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra“. In: *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Hrsg. von Francesco Logozzo und Manuel Fähndrich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 345–365.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3. Aufl. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[FPS18] Adrian Francalanza, Jorge A. Perez und César Sánchez. „Runtime Verification for Decentralized and Distributed Systems“. In: *Lectures on Runtime Verification – Introductory and Advanced Topics*. Bd. 10457. LNCS. Springer, 2018. Kap. 6, S. 169–205.

[Fra+13] Pierre Fraigniaud u. a. „What can be decided locally without identifiers?“ In: *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. Hrsg. von Panagiota Fatourou und Gadi Taubenfeld. ACM, 2013, S. 157–165.

[Fra+17] Adrian Francalanza u. a. „A Foundation for Runtime Monitoring“. In: Bd. 10548. Sep. 2017, S. 8–29.

[Fra80] Nissim Francez. „Distributed Termination“. In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), S. 42–55.

[FS16] Yliès Falcone und César Sánchez, Hrsg. *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*. Cham: Springer International Publishing, 2016.

[Geu09] H. Geuvers. „Proof assistants: History, ideas and future“. In: *SADHANA - Academy Proceedings in Engineering Sciences* 34 (2009), S. 3–25.

[GGP10] Rosario Gennaro, Craig Gentry und Bryan Parno. „Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers“. In: *Proceedings of the 30th Annual Conference on Advances in Cryptology*. CRYPTO'10. Santa Barbara, CA, USA: Springer-Verlag, 2010, S. 465–482.

[Gho14] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach, Second Edition*. 2nd. Chapman & Hall/CRC, 2014.

[Gie+19] Holger Giese u. a. „Metric Temporal Graph Logic over Typed Attributed Graphs“. In: *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Hrsg. von Reiner Hähnle und Wil M. P. van der Aalst. Bd. 11424. Lecture Notes in Computer Science. Springer, 2019, S. 282–298.

[Gle03] Sabine Glesner. „Program Checking with Certificates: Separating Correctness-Critical Code“. English. In: *FME 2003: Formal Methods*. Hrsg. von Keijiro Araki, Stefania Gnesi und Dino Mandrioli. Bd. 2805. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 758–777.

[Gol10] Oded Goldreich. *P, NP, and NP-Completeness: The Basics of Computational Complexity*. 1st. USA: Cambridge University Press, 2010.

[Gon08] Georges Gonthier. „The Four Colour Theorem: Engineering of a Formal Proof“. In: *Computer Mathematics*. Hrsg. von Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 333–333.

[GRS18] Seth Gilbert, Peter Robinson und Suman Sourav. „Leader Election in Well-Connected Graphs“. In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. PODC '18. Egham, United Kingdom: ACM, 2018, S. 227–236.

[GS07] Michael T. Goodrich und Jonathan Z. Sun. „Checking Value-sensitive Data Structures in Sublinear Space“. In: *Proceedings of the 18th International Conference on Algorithms and Computation*. ISAAC'07. Sendai, Japan: Springer-Verlag, 2007, S. 353–364.

[GS11] Mika Göös und Jukka Suomela. „Locally checkable proofs“. In: *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*. Hrsg. von Cyril Gavoille und Pierre Fraigniaud. ACM, 2011, S. 159–168.

[GTW06] Benjamin Grégoire, Laurent Théry und Benjamin Werner. „A Computational Approach to Pocklington Certificates in Type Theory“. In: *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006. Proceedings*. Hrsg. von Masami Hagiya und Philip Wadler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 97–113.

[Gur95] Yuri Gurevich. „Specification and Validation Methods“. In: Hrsg. von Egon Börger. New York, NY, USA: Oxford University Press, Inc., 1995. Kap. Evolving Algebras 1993: Lipari Guide, S. 9–36.

[HA11] Maaly Hassan und Chickadel Andrew. „A Review of Interference Reduction in Wireless Networks Using Graph Coloring Methods“. In: *International Journal on Applications of Graph Theory in Wireless ad hoc Networks and Sensor Networks* 3 (März 2011).

[Hal16] Sylvain Hallé. „When RV Meets CEP“. In: *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*. Hrsg. von Yliès Falcone und César Sánchez. Cham: Springer International Publishing, 2016, S. 68–91.

[HC11] Ruo-Wei Hung und Maw-Shang Chang. „An efficient certifying algorithm for the Hamiltonian cycle problem on circular-arc graphs“. In: *Theoretical Computer Science* 412.39 (2011), S. 5351 –5373.

[Hel+12] Lauri Hella u. a. „Weak Models of Distributed Computing, with Connections to Modal Logic“. In: *CoRR* abs/1205.2051 (2012).

[HH05] Pavol Hell und Jing Huang. „Certifying LexBFS Recognition Algorithms for Proper Interval Graphs and Proper Interval Bigraphs“. In: *SIAM Journal on Discrete Mathematics* 18 (3 2005), S. 554–570.

[HK07] Pinar Heggernes und Dieter Kratsch. „Linear-time Certifying Recognition Algorithms and Forbidden Induced Subgraphs“. In: *Nordic Journal of Computing* 14.1 (Jan. 2007), S. 87–108.

[HN19] A.E. Hodler und M. Needham. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media, 2019.

[HNR16] Martin Hofmann, Christian Neukirchen und Harald Rueß. „Certification for μ calculus with winning strategies“. In: *Model Checking Software: 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. Hrsg. von Dragan Bošnački und Anton Wijs. Cham: Springer International Publishing, 2016, S. 111–128.

[Hol] *Homepage des Beweisassistenten HOL Light*. http://www.cl.cam.ac.uk/~jrh13/hol-light/.

[Hol+16] Sönke Holthusen u. a. „Proof-Carrying Apps: Contract-Based Deployment-Time Verification“. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Cham: Springer International Publishing, 2016, S. 839–855.

[Hug02] Marc-Philippe Huget. „Extending Agent UML Sequence Diagrams“. In: *Proceedings of the 3rd International Conference on Agent-Oriented Software Engineering III*. AOSE'02. Bologna, Italy: Springer-Verlag, 2002, S. 150–161.

[HZW12] Chu Huang, Sencun Zhu und Dinghao Wu. „Towards Trusted Services: Result Verification Schemes for MapReduce“. In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, S. 41–48.

[Ins] *InSystems – proANT Transport Robots*. http://www.insystems.de/en/produkte/proant-transport-roboter/.

[Isa] *Homepage des Beweisassistenten Isabelle*. http://isabelle.in.tum.de/.

[Jak17] Marie-Christine Jakobs. „On-The-Fly safety checking – customizing program certification and program restructuring“. Diss. University of Paderborn, 2017.

[Joh+13] Annu John u. a. „Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms“. In: *SPIN*. Bd. 7976. LNCS. 2013, S. 209–226.

[JTL12] Dongseok Jang, Zachary Tatlock und Sorin Lerner. „Establishing Browser Security Guarantees through Formal Shim Verification“. In: *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association. 2012, S. 8.

[JW17] Marie-Christine Jakobs und Heike Wehrheim. „Programs from Proofs: A Framework for the Safe Execution of Untrusted Software“. In: *ACM Trans. Program. Lang. Syst.* 39.2 (März 2017), 7:1–7:56.

[KH13] Safwan Mahmud Khan und Kevin W. Hamlen. „Computation Certification as a Service in the Cloud“. In: *2013 13th IEEE/ACM International Symposium onCluster, Cloud and Grid Computing (CCGrid), Delft Netherlands* (Mai 2013), S. 434–441.

[KK06] Amos Korman und Shay Kutten. „Distributed Verification of Minimum Spanning Trees“. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '06. Denver, Colorado, USA: ACM, 2006, S. 26–34.

[KKP10] Amos Korman, Shay Kutten und David Peleg. „Proof labeling schemes“. In: *Distributed Computing* 22.4 (2010), S. 215–233.

[KLO10] Fabian Kuhn, Nancy Lynch und Rotem Oshman. „Distributed Computation in Dynamic Networks“. In: *Proceedings of the Forty-second ACM Symposium on Theory of Computing*. STOC '10. Cambridge, Massachusetts, USA: ACM, 2010, S. 513–522.

[KMW04] Fabian Kuhn, Thomas Moscibroda und Rogert Wattenhofer. „What Cannot Be Computed Locally!“ In: *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*. PODC '04. St. John's, Newfoundland, Canada: ACM, 2004, S. 300–309.

[KN09] Haim Kaplan und Yahav Nussbaum. „Certifying algorithms for recognizing proper circular-arc graphs and unit circular-arc graphs“. In: *Discrete Applied Mathematics* 157.15 (2009), S. 3216 –3230.

[KNR12] Philipp Küfner, Uwe Nestmann und Christina Rickmann. „Formal Verification of Distributed Algorithms“. In: *Theoretical Computer Science: 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*. Hrsg. von Jos C. M. Baeten, Tom Ball und Frank S. de Boer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 209–224.

[KS18] Narges Khakpour und Charilaos Skandylas. „Synthesis of a Permissive Security Monitor“. In: *Computer Security*. Hrsg. von Javier Lopez, Jianying Zhou und Miguel Soriano. Springer International Publishing, 2018, S. 48–65.

[Leg] *Homepage des Beweisassistenten LEGO*. http://www.dcs.ed.ac.uk/home/lego/.

[Les11] S. Lescuyer. „Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq“. Diss. Université de Paris, 2011.

[Leto8] Pierre Letouzey. „Extraction in Coq: An Overview“. In: *Logic and Theory of Algorithms*. Hrsg. von Arnold Beckmann, Costas Dimitracopoulos und Benedikt Löwe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 359–369.

[LH93] M. R. Lyu und Y. . He. „Improving the N-version programming process through the evolution of a design paradigm“. In: *IEEE Transactions on Reliability* 42.2 (1993), S. 179–189.

[LMo7] Jinyuan Li und David Maziéres. „Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems“. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI'07. Cambridge, MA: USENIX Association, 2007, S. 10–10.

[LR17] Shuvendu K. Lahiri und Giles Reger, Hrsg. *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. Bd. 10548. Lecture Notes in Computer Science. Springer, 2017.

[LSL17] Yanhong Liu, Scott Stoller und Bo Lin. „From Clarity to Efficiency for Distributed Algorithms“. In: *ACM Transactions on Programming Languages and Systems* 39 (Mai 2017), S. 1–41.

[LSP82] Leslie Lamport, Robert Shostak und Marshall Pease. „The Byzantine Generals Problem“. In: *ACM Trans. Program. Lang. Syst.* 4.3 (Juli 1982), S. 382–401.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[MB15] Menna Mostafa und Borzoo Bonakdarpour. „Decentralized Runtime Verification of LTL Specifications in Distributed Systems“. In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IPDPS '15. Washington, DC, USA: IEEE Computer Society, 2015, S. 494–503.

[McC+11] Ross M. McConnell u. a. „Certifying Algorithms“. In: *Computer Science Review* 5 (2 2011), S. 119–161.

[McC04] Ross M. McConnell. „A Certifying Algorithm for the Consecutive-ones Property“. In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '04. New Orleans, Louisiana: Society for Industrial und Applied Mathematics, 2004, S. 768–777.

[Med10] D. Medhi. *Network Routing: Algorithms, Protocols, and Architectures*. The Morgan Kaufmann Series in Networking. Elsevier Science, 2010.

[Min] *Homepage des Beweisassistenten MINLOG*. http://www.mathematik.uni-muenchen.de/~logik/minlog/.

[Miz] *Homepage des Beweisassistenten Mizar*. http://mizar.uwb.edu.pl/.

[ML06] Per Martin-Löf. „100 Years of Zermelo's Axiom of Choice: What Was the Problem with It?“ In: *Comput. J.* 49.3 (Mai 2006), S. 345–350.

[MN98] Kurt Mehlhorn und Stefan Näher. „From Algorithms to Working Programs: On the Use of Program Checking in LEDA“. In: *Lecture Notes in Computer Science* 1450 (1998), S. 84–93.

[Mos11] Michał Moskal. „Verifying Functional Correctness of C Programs with VCC“. In: *Proceedings of the Third International Conference on NASA Formal Methods*. NFM'11. Pasadena, CA: Springer-Verlag, 2011, S. 56–57.

[MP16] Stefan Mitsch und André Platzer. „ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models“. In: *Form. Methods Syst. Des.* 49 (Feb. 2016).

[MSF11] Mircea Moca, Gheorge Cosmin Silaghi und Gilles Fedak. „Distributed Results Checking for MapReduce in Volunteer Computing“. In: *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, Shanghai China*. IEEE, 2011.

[Mul+18] Eric Mullen u. a. „Oeuf: Minimizing the Coq Extraction TCB“. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, S. 172–185.

[Nam01] Kedar S. Namjoshi. „Certifying Model Checkers“. In: *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*. Hrsg. von Gérard Berry, Hubert Comon und Alain Finkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 2–13.

[NCM14] Martin Nordio, Cristiano Calcagno und Bertrand Meyer. „Certificates and Separation Logic“. In: *8th International Symposium on Trustworthy Global Computing - Volume 8358*. TGC 2013. Buenos Aires, Argentina: Springer-Verlag New York, Inc., 2014, S. 273–293.

[Nec97] George C. Necula. „Proof-carrying Code“. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: ACM, 1997, S. 106–119.

[NL00] George C. Necula und Peter Lee. „Proof Generation in the Touchstone Theorem Prover“. In: *Automated Deduction - CADE-17*. Hrsg. von David McAllester. Bd. 1831. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 25–44.

[NL98] George C. Necula und Peter Lee. „The Design and Implementation of a Certifying Compiler“. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, S. 333–344.

[Nos14] Lars Noschinski. „A Graph Library for Isabelle“. In: *Mathematics in Computer Science* 9 (März 2014), S. 23–39.

[NP05] Hans de Nivelle und Ruzica Piskac. „Verification of an Off-Line Checker for Priority Queues“. In: *SEFM*. IEEE Computer Society, 2005, S. 210–219.

[NP12] Stavros D. Nikolopoulos und Leonidas Palios. „An O(nm)-time certifying algorithm for recognizing HHD-free graphs“. In: *Theoretical Computer Science* 452 (2012), S. 117 –131.

[NRM14] Lars Noschinski, Christine Rizkallah und Kurt Mehlhorn. „Verification of Certifying Computations through AutoCorres and Simpl“. English. In: *NASA Formal Methods*. Hrsg. von Julia M. Badger und Kristin Yvonne Rozier. Bd. 8430. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 46–61.

[NS95] Moni Naor und Larry Stockmeyer. „What Can Be Computed Locally?“ In: *SIAM J. Comput.* 24.6 (Dez. 1995), S. 1259–1277.

[NSH14] T.M. Ngo, M. Stoelinga und M. Huisman. „Effective Verification of Confidentiality for Multi-threaded Programs“. In: *Journal of Computer Security* 22 (2014), S. 269–300.

[Nup] *Homepage des Beweisassistenten NuPRL*. http://www.nuprl.org/.

[OPR17] Rafail Ostrovsky, Mor Perry und Will Rosenbaum. „Space-Time Tradeoffs for Distributed Verification“. In: *Structural Information and Communication Complexity*. Hrsg. von Shantanu Das und Sebastien Tixeuil. Cham: Springer International Publishing, 2017, S. 53–70.

[Peloo] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Philadelphia, PA, USA: Society for Industrial und Applied Mathematics, 2000.

[PM12] Christine Paulin-Mohring. „Introduction to the Coq Proof-Assistant for Practical Software Verification“. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Hrsg. von Bertrand Meyer und Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 45–95.

[PM15] Christine Paulin-Mohring. „Introduction to the Calculus of Inductive Constructions“. In: *All about Proofs, Proofs for All*. Hrsg. von Bruno Woltzenlogel Paleo und David Delahaye. Bd. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015.

[Pot15] François Pottier. „Depth-First Search and Strong Connectivity in Coq“. In: (Jan. 2015).

[Ray+11] Arnab Ray u. a. „Functional and Nonfunctional Design Verification for Embedded Software Systems“. In: *Advances in Computers* 83 (2011), S. 277–321.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Berlin Heidelberg, 2013.

[Rei10] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. 1st. Springer Publishing Company, Incorporated, 2010.

[Ric+14] Daniel Ricketts u. a. „Automating Formal Proofs for Reactive Systems“. In: *ACM SIGPLAN Notices*. Bd. 49. 6. ACM. 2014, S. 452–462.

[Riz14] Christine Rizkallah. „A Simpl Shortest Path Checker Verification“. eng. In: *Proceedings of Isabelle Workshop 2014*. 2014.

[Riz15] Christine Rizkallah. „Verification of Program Computations“. Diss. Saarland University, 2015.

[Rub90] Ronitt Rubinfeld. „A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs“. Diss. University of California, Berkeley, 1990.

[San06] Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. USA: Wiley-Interscience, 2006.

[Sch12] Jens M. Schmidt. „Construction Sequences and Certifying 3-Connectivity“. In: *Algorithmica* 62 (2012), S. 192–208.

[Sch18] H. Schlingloff. „Specification and Verification of Collaborative Transport Robots“. In: *2018 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC)*. 2018, S. 3–8.

[SM90] Gregory F. Sullivan und Gerald M. Masson. „Using Certification Trails to Achieve Software Fault Tolerance“. In: *Proceedings of IEEE 20th International Symposium on Fault-Tolerant Computing (FTCS)*. 1990, S. 423–431.

[SM91] G. F. Sullivan und G. M. Masson. „Certification Trails for Data Structures“. In: *Digest of the 21st Symposium on Fault-Tolerant Computing* (1991), S. 240–247.

[SS03] Natarajan Shankar und Maria Sorea. *Counterexample-Driven Model Checking*. 2003.

[Suo13] Jukka Suomela. „Survey of Local Algorithms“. In: *ACM Comput. Surv.* 45.2 (März 2013), 24:1–24:40.

[SWM93] Gregory F. Sullivan, Dwight S. Wilson und Gerald M. Masson. „Certification Trails and Software Design for Testability“. In: *Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics - Join Them*. Washington, DC, USA: IEEE Computer Society, 1993, S. 200–209.

[SWM95] Gregory F. Sullivan, Dwight S. Wilson und Gerald M. Masson. „Certification of Computational Results“. In: *IEEE Transactions on Computers*. IEEE, 1995, S. 833–847.

[SWT17] Ilya Sergey, James R. Wilcox und Zachary Tatlock. „Programming and Proving with Distributed Protocols“. In: *Proc. ACM Program. Lang.* 2.POPL (Dez. 2017).

[SYY03] Sunae Seo, Hongseok Yang und Kwangkeun Yi. „Automatic Construction of Hoare Proofs from Abstract Interpretation Results“. In: *Programming Languages and Systems: First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003. Proceedings*. Hrsg. von Atsushi Ohori. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, S. 230–245.

[Tar79] Robert Endre Tarjan. „Applications of Path Compression on Balanced Trees“. In: *J. ACM* 26.4 (Okt. 1979), S. 690–715.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. New York, NY, USA: Cambridge University Press, 1994.

[Tho84] Ken Thompson. „Reflections on Trusting Trust“. In: *Commun. ACM* 27.8 (Aug. 1984), S. 761–763.

[VA17] Kim Völlinger und Samira Akili. „Verifying a Class of Certifying Distributed Programs“. In: *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*. Hrsg. von Clark Barrett, Misty Davies und Temesghen Kahsai. Bd. 10227. Lecture Notes in Computer Science. 2017, S. 373–388.

[VA18] Kim Völlinger und Samira Akili. „On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency“. In: *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*. Hrsg. von Christel Baier und Luís Caires. Bd. 10854. Lecture Notes in Computer Science. Springer, 2018, S. 161–180.

[Var10] Andras Varga. „OMNeT++“. In: *Modeling and Tools for Network Simulation*. Hrsg. von Klaus Wehrle, Mesut Güneş und James Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 35–59.

[Var93] George Varghese. „Self-stabilization by Local Checking and Correction“. Diss. Cambridge, MA, USA, 1993.

[VR15] Kim Völlinger und Wolfgang Reisig. „Certification of Distributed Algorithms Solving Problems with Optimal Substructure“. In: *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. Hrsg. von Radu Calinescu und Bernhard Rumpe. Bd. 9276. Lecture Notes in Computer Science. Springer, 2015, S. 190–195.

[Völ17] Kim Völlinger. „Verifying the Output of a Distributed Algorithm Using Certification“. In: *Runtime Verification: 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. Hrsg. von Shuvendu Lahiri und Giles Reger. Cham: Springer International Publishing, 2017, S. 424–430.

[Völ19] Kim Völlinger. „On Certifying Distributed Algorithms: Problem of Local Correctness“. In: *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*. Hrsg. von Jorge A. Pérez und Nobuko Yoshida. Bd. 11535. Lecture Notes in Computer Science. Springer, 2019, S. 281–288.

[Vö13] Kim Völlinger. *Einsatz des Beweisassistenten COQ zur deduktiven Programmverifikation*. Humboldt-Universität zu Berlin. 2013.

[Wad15] Philip Wadler. „Proposition as Types“. In: *Commun. ACM* 58.12 (2015).

[Wad95] Philip Wadler. „Monads for Functional Programming“. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Berlin, Heidelberg: Springer-Verlag, 1995, S. 24–52.

[WAG12] Iain Whiteside, David Aspinall und Gudmund Grov. „An Essence of SSReflect“. In: *Proceedings of the 11th International Conference on Intelligent Computer Mathematics*. CICM 12. Bremen, Germany: Springer-Verlag, 2012, S. 186–201.

[WB15] Michael Walfish und Andrew J. Blumberg. „Verifying Computations Without Reexecuting Them“. In: *Commun. ACM* 58.2 (Jan. 2015), S. 74–84.

[Whi11] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, Inc., 2011.

[Wil+04] Martin Wildmoser u. a. „Prototyping Proof Carrying Code“. In: *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France*. Hrsg. von Jean-Jacques Levy, Ernst W. Mayr und John C. Mitchell. Boston, MA: Springer US, 2004, S. 333–347.

[Wil+15a] James R. Wilcox u. a. „Verdi: A framework for implementing and formally verifying distributed systems“. In: *PLDI 2015: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*. Portland, OR, USA, Juni 2015, S. 357–368.

[Wil+15b] James R Wilcox u. a. „Verdi: A Framework for Implementing and Formally Verifying Distributed Systems“. In: *ACM SIGPLAN Notices*. Bd. 50. 6. ACM. 2015, S. 357–368.

[WMM10] Ryan Wisnesky, Gregory Michael Malecha und John Gregory Morrisett. „Certified Web Services in Ynot“. In: *Workshop on Automated Specification and Verification of Web Systems* (2010).

[Woo+16] Doug Woos u. a. „Planning for Change in a Formal Verification of the Raft Consensus Protocol“. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM. 2016, S. 154–165.

[XL10] Shuguang Xiong und Jianzhong Li. „An Efficient Algorithm for Cut Vertex Detection in Wireless Sensor Networks“. In: *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*. ICDCS '10. Washington, DC, USA: IEEE Computer Society, 2010, S. 368–377.

[ZPK14] Yupeng Zhang, Charalampos Papamanthou und Jonathan Katz. „ALITHEIA: Towards Practical Verifiable Graph Processing“. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, S. 856–867.

[Abd+14] L. M. Abdullah u. a. „New graph colouring algorithm for resource allocation in large-scale wireless networks“. In: *2014 IEEE 5th Control and System Graduate Research Colloquium*. 2014, S. 233–238.

[Ain15] Claire Ainsworth. „Sex redefined“. In: *Nature* 518.7539 (2015), S. 288–291.

[Shi+17] H. Shi u. a. „Monitor Synthesis for Parametric MTL Properties in Discrete Control Software“. In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2017, S. 355–362.

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich erkläre, dass ich die Dissertation selbständig und nur unter Verwendung der von mir gemäß §7 Abs. 3 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42/2018 am 11.07.2018 angegebenen Hilfsmittel angefertigt habe.

Verteidigt am 8.6.2020

Kim Völlinger